André Filipe Pereira Ramos

Licenciado em Engenharia Informática

IDE baseado em Eclipse para CxProlog

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador: Artur Miguel Dias, Prof.Auxiliar,
Universidade Nova de Lisboa

Júri:

Presidente: Pedro Barahona, Prof.Catedrático
Universidade Nova de Lisboa

Arguente: Salvador Abreu, Prof.Catedrático
Universidade de Évora

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2016

IDE baseado em Eclipse para CxProlog

Dedicatória

Para toda a minha família, que sempre esteve ao meu lado nos bons e maus momentos, sempre me suportou e apoiou. Sem eles nunca conseguiria ter chegado ao momento que me levou a redigir este documento.

Para todos os meus amigos que não só revelaram uma total disponibilidade para ajudar como providenciaram preciosos momentos de diversão que aliviaram a tensão de difíceis momentos.

Para todos os meus Professores, ao longo de toda a minha vida escolar. Os conhecimentos que me transmitiram permitiram-me alcançar os meus objetivos.

Agradecimentos

Quero agradecer ao meu orientador – Professor Artur Miguel Dias. Por a tremenda disponibilidade, os importantes conselhos e pelo olho clínico na revisão de todo o documento.

Quero agradecer a toda a equipa da SQIMI com quem tive contacto, nomeadamente: Tiago, Rosângela, Nuno e Susana que revelaram uma total disponibilidade para ajudar durante todo o projeto.

Resumo

O CxProlog é uma implementação da linguagem Prolog suportando várias extensões e desenvolvida no NOVA LINCS do Departamento de Informática, FCT, Universidade Nova de Lisboa sendo o seu criador o Professor A.Miguel Dias.

O projeto desta dissertação consiste na conceção e implementação dum IDE baseado em Eclipse para o CxProlog. O IDE destina-se a suportar a escrita, teste e execução de programas CxProlog. Deverá disponibilizar as funcionalidades mais características dos IDEs atuais (realce de sintaxe, completação automática, etc.), assim como outras funcionalidades que sejam relevantes para um programador de CxProlog (ex: injeção de golos na consola).

Uma das maiores ambições do projeto é que o IDE seja configurável e parcialmente definido utilizando a própria linguagem CxProlog. Tal ambição requer a utilização de mecanismos de interoperabilidade entre CxProlog e Java (a linguagem que implementa o Eclipse).

Os maiores desafios do projeto são o domínio da plataforma Eclipse, principalmente os seus mecanismos para desenvolvimento de *plugins*, a interoperabilidade entre CxProlog e Java/Eclipse, e a implementação das funcionalidades mais sofisticadas.

O documento apresentado engloba: planeamento, implementação e utilização prática do projeto.

Para além disso o utilizador também terá acesso ao estudo realizado sobre o estado de arte dos IDEs e à avaliação do sistema.

Palavras-chave: *IDE, Eclipse, CxProlog, Plugins, Interoperabilidade*

Abstract

CxProlog is an implementation of the Prolog language with extensions, developed in NOVA LINCS of Departamento de Informática, FCT, Universidade Nova de Lisboa being its creator the Professor A.Miguel Dias.

The project of this dissertation consists of the design and implementation of an Eclipse-based IDE for CxProlog. The IDE is intended to support the writing, testing and execution of CxProlog programs. It should provide the most characteristic features of current IDEs (syntax highlight, auto-completion, etc.), and other features that benefit a CxProlog programmer (ex: injection of goals in the console).

One of the greatest ambitions of the project is that the IDE be configurable and partially set using the CxProlog language itself. This ambition requires the use of mechanism of interoperability between CxProlog and Java (the language that implements Eclipse).

The biggest challenges of the projet are: mastering the Eclipse platform, especially its mechanisms for plugins development, the interoperability between CxProlog and Eclipse/Java, and the implementation of the more sophisticated features.

The presented document includes: planning, implementation and practical use of the project.

In addition, the reader also has acess to the study on the state of art of the IDEs and the system evaluation.

Keywords: *IDE, Eclipse, CxProlog, Plugin, Interoperability*

Índice

Lista de figuras

Lista de Tabelas

1 Introdução

Um ambiente de desenvolvimento integrado (IDE) engloba todas as funcionalidades e ferramentas necessárias para o desenvolvimento de *software* num único programa e possibilita a interação com o sistema através duma interface comum. Os IDEs são desenhados de modo a maximizar a produtividade dos seus utilizadores e atualmente cerca de 97% dos programadores de *software* usam IDEs (1). Um bom IDE pode fazer a diferença entre uma boa ou má experiência de programação. De facto, as linguagens de programação mais utilizadas atualmente, Java e C, são suportadas por IDEs bastante populares e completos oferecidos pelo Eclipse e Microsoft Visual Studio. A qualidade duma linguagem de programação têm grande importância para a sua aceitação por parte da comunidade, mas os IDEs que a suportam podem ser um fator importante para a sua valorização.

O projeto apresentado consiste na conceção e implementação dum IDE baseado em Eclipse para o CxProlog. O IDE destina-se a suportar a escrita, teste e execução de programas em CxProlog. Além disso, o IDE deverá disponibilizar as funcionalidades mais características dos IDEs atuais (realce de sintaxe, completação automática, consola, etc.), assim como outras funcionalidades que sejam relevantes para um programador de CxProlog.

Não era ambição criar um IDE de raiz, sendo escolhido o Eclipse como plataforma para o desenvolvimento do IDE. A opção pelo Eclipse deveu-se à sua grande notoriedade (é o IDE mais utilizado, por cerca de 30% dos desenvolvedores de *software* (2)) e à sua arquitetura que facilita a extensibilidade.

Uma das maiores ambições do projeto é que o IDE seja configurável e parcialmente definido utilizando a própria linguagem CxProlog. Isto é, o utilizador poderá estender o IDE em Eclipse através do CxProlog e não do Java (linguagem que implementa o Eclipse). A anterior ambição requer a utilização de mecanismos de interoperabilidade entre CxProlog e Java.

Os maiores desafios a superar neste projeto são: o domínio da plataforma Eclipse, principalmente os seus mecanismos para desenvolvimento de *plugins*, a interoperabilidade entre CxProlog e Eclipse/Java e a implementação das funcionalidades mais sofisticadas.

CxIDE foi o nome escolhido para o IDE deste projeto. O presente documento engloba a descrição detalhada de todas as funcionalidades do CxIDE, aos seguintes três níveis: importância, implementação e utilização prática. O leitor terá igualmente acesso ao estudo realizado sobre o estado de arte dos IDEs em geral e para o Prolog em particular, planeamento inicial de todo o projeto, e avaliação crítica do mesmo.

1.1 Motivação

O sucesso duma linguagem de programação moderna é influenciado não só pelas características da linguagem, mas também pela qualidade das ferramentas disponíveis para a mesma: compiladores, depuradores, editores, etc. Modernamente, existe a tendência para integrar as funcionalidades das várias ferramentas numa aplicação única, o IDE. Um programador escreve, documenta e testa os seus programas dentro dum IDE e é lá que passa a maior parte do seu tempo de trabalho (3).

Depois do seu aparecimento, os IDEs sofreram uma rápida evolução tornando-se alguns deles muito completos e sofisticados. Modernamente, ainda se verifica alguma inovação nos IDEs de forma a dar resposta às atuais necessidades dos programadores. Eis alguns exemplos das mais recentes inovações:

- IDEs Cloud – São IDEs em que todos os ficheiros do projeto se encontram no repositório da *cloud* e os programadores trabalham colaborativamente através da Net (4).
- IDEs Formais – São IDEs muito avançados que usam técnicas de demonstração de programas para validar determinadas propriedades como a correção e a segurança (5).
- IDEs que aproveitam o conhecimento das comunidades - Recentemente têm sido propostas soluções que possibilitam ao utilizador encontrar respostas às suas dificuldades de programação sem recorrer aos navegadores web (6) (7).

O Eclipse e Microsoft Visual Studio disponibilizam alguns dos mais populares IDEs. Esses IDEs suportam linguagens como Java e C# (entre outras) (8) (9), e podem ser considerados como o expoente máximo dos IDEs atuais, ao oferecerem um grande leque de funcionalidades e ao estarem suportados por uma grande comunidade. Também integram ferramentas como: compilador, construtor e depurador.

Os IDEs não garantem um aumento de qualidade no código, mas facilitam e agilizam o processo de desenvolvimento incrementando a produtividade do programador (1). A produtividade é um dos fatores mais importantes para o sucesso dum programador, e assim a utilização de IDEs é atualmente algo quase indispensável.

Nem sempre os programadores puderam usufruir da utilização de IDEs. Por exemplo, nos primórdios do C, não existiam IDEs. Os programadores editavam os programas em simples editores de texto e compilavam os programas escrevendo comandos num terminal. Porém hoje em dia cerca de 97% dos programadores não prescinde da utilização de IDEs (1), sendo relativamente rara a utilização de compiladores fora do contexto dum IDE. Atualmente, os programadores têm a expetativa de usar as linguagens de programação no contexto de IDEs. Num mundo cada vez mais competitivo a nível empresarial, é imprescindível a utilização dum IDE de forma aumentar a não só a produtividade como também o conforto dos programadores.

O Eclipse é uma plataforma para o desenvolvimento de *software* que foi construída desde o seu início de modo a facilitar a extensibilidade (10). Tal facilidade de extensão permitiu que vários IDEs de grande qualidade fossem desenvolvidos para o Eclipse (11), ao ponto de que qualquer das linguagens mais utilizadas no mundo tem um IDE baseado em Eclipse.

Mas de todas as motivações, a maior, é o facto de que um IDE para CxProlog seria uma mais-valia para todos os programados de CxProlog (utilizado a nível profissional pela SQIMI[1]), que atualmente usam apenas editores de texto (ex: Joe[2]) para edição de programas e terminal para a sua execução.

[1] Mais informações sobre a SQIMI em: www.sqimi.com
[2] Mais informações sobre o editor Joe em: pt.wikipedia.org/wiki/Joe_(editor_de_texto)

1.2 Conceitos orientadores do CxIDE

São destacados quatro importantes conceitos relativos ao CxIDE. Foram estes os conceitos essenciais que orientaram todo o esforço de desenho do IDE. Esta secção pretende ajudar o leitor a contextualizar tudo o que será descrito nos posteriores capítulos.

1º - Imitação

O CxIDE deve ser funcionalmente semelhante a outros IDEs baseados em Eclipse (por exemplo Java IDE ou C/C++ IDE), de forma a proporcionar um ambiente familiar ao habitual utilizador do Eclipse.

Sendo o IDE configurável e extensível, o utilizador também pode reconfigurar o sistema violando o princípio da imitação, mas fará isso à sua responsabilidade. Por exemplo, o utilizador poderia remover todos os menus do Eclipse (*File*, *Edit*, *etc.*) através do CxIDE, tal feito causaria confusão a um outro utilizador que esteja habituado ao *layout* padrão do Eclipse.

2º - Extensibilidade usando CxProlog

O utilizador sofisticado deve poder estender o seu IDE com novas funcionalidades integralmente programadas em CxProlog. Geralmente essas funcionalidades ficam acessíveis através de menus (convencionais ou de contexto (*popup*)) que o próprio utilizador instala usando código CxProlog.

3º - Configurabilidade usando CxProlog

O utilizador sofisticado deve poder configurar as funcionalidades nucleares (implementadas em Java) e as funcionalidades adicionais (implementadas em CxProlog) escrevendo código CxProlog num ficheiro de arranque. Por exemplo: mudar as regras de realce de sintaxe do editor, eliminar e adicionar menus, criar diálogos, etc.

4º - Ambiente de execução de programas CxProlog

Possibilidade de desenvolver no CxIDE dois tipos de programa CxProlog: programas autónomos que irão correr fora do CxIDE; ou programas que se destinam a correr exclusivamente dentro do CxIDE e que aproveitam a parte gráfica do Eclipse (instalando novos menus, criando novos diálogos, etc.). O segundo estilo de utilização mostra que o CxIDE pode ser também considerado um ambiente de execução de programas CxProlog, a fazer lembrar o clássico ambiente de desenvolvimento e execução do *SmallTalk*[3].

[3]Mais informações sobre o Smalltalk: pt.wikipedia.org/wiki/Smalltalk

1.3 Organização da dissertação

O documento está dividido em dez capítulos principais:

1. **Introdução (Capítulo 1)**
   Apresentação do projeto proposto e motivação para o mesmo.

2. **Ambientes de programação e IDEs (Capítulo 2)**
   Definição de conceitos importantes para a compreensão do projeto. Informações sobre os IDES mais utilizados modernamente.

3. **Prolog e seus IDEs (Capítulo 3)**
   Descrição do CxProlog. Informações sobre IDEs para Prolog.

4. **Funcionalidades e ferramentas dos IDEs (Capítulo 4)**
   Enumeração e descrição de várias funcionalidades e ferramentas que os IDEs tendem a disponibilizar.

5. **Estender o Eclipse (Capítulo 5)**
   Exemplo prático de como criar um *plugin* no Eclipse.

6. **Planeamento do CxIDE (Capítulo 6)**
   Explicação de algumas decisões iniciais e enumeração de objetivos do projeto.

7. **Funcionalidades do CxIDE (Capítulo 7)**
   Descrição detalhada de todas as funcionalidades que o projeto final inclui.

8. **Implementação (Capítulo 8)**
   Descrição detalhada da implementação do projeto.

9. **Avaliação do sistema (Capítulo 9)**
   Testes de utilizadores e benchmarks.

10. **Conclusão (Capítulo 10)**

    Conclusões sobre o projeto final e trabalho futuro.

2 Ambientes de programação e IDEs

2.1 Ambientes de programação e ambientes de desenvolvimento integrado

Um ambiente de programação é uma coleção de ferramentas usadas no desenvolvimento de s*oftware*. Um bom sistema de desenvolvimento aumenta a produtividade dos programadores facilitando e acelerando o desenvolvimento de *software*. As funcionalidades estão coordenadas num todo coerente, de acordo com uma determinada visão do processo de desenvolvimento de *software*. Por exemplo, o ambiente de desenvolvimento não visual JDK[4] (*Java Development Kit*) consiste num conjunto de ferramentas que ajudam a desenvolver, testar, documentar e executar programas em Java. Estas ferramentas podem ser usadas na linha de comandos, mas também podem ser integradas em ambientes de desenvolvimento gráfico, como por exemplo o Eclipse[5].

Um IDE (*Integrated Development Environment*), ou ambiente de desenvolvimento integrado é um ambiente de programação especial. O IDE é um programa único dentro do qual se processa todo o desenvolvimento de *software*. Geralmente a interface é gráfica. Os IDEs com uma arquitetura mais sofisticada são extensíveis através de *plugins* ou conceitos semelhantes.

Uma importante plataforma para desenvolver IDEs usando *plugins* é o Eclipse. Mais informações sobre IDEs e *plugins* serão apresentadas posteriormente.

2.2 IDEs mais utilizados

Em 2015, o IDE mais utilizado era o Eclipse, seguido por perto pelo Visual Studio (2). Nesta secção serão apresentadas informações sobre alguns dos IDEs mais utilizados, designadamente: Eclipse, Visual Studio e NetBeans. Será dado um especial destaque ao Eclipse, revelando a sua arquitetura e realizando algumas comparações com os restantes IDEs apresentados.

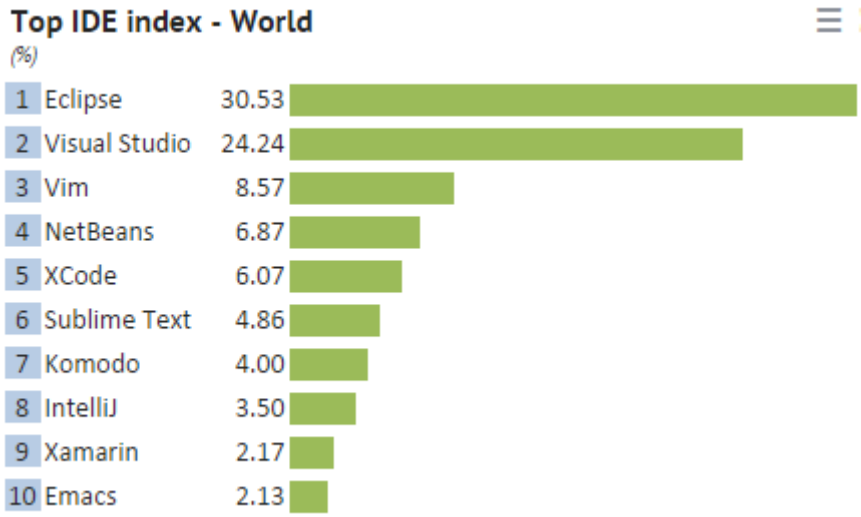


Figura 1 – Os IDEs mais utilizados

[4] Mais informações sobre o JDK em: goo.gl/ftzj0k
[5] Mais informações sobre o Eclipse em: eclipse.org

É importante realçar que o Vim, Emacs e outros presentes na figura 1 representam uma discussão que dura há vários anos. O assunto discutido é se certos editores de texto (como o Vim) podem ser considerados IDEs ou não. A maioria da comunidade defende que as versões padrão desses editores de texto não são IDEs, mas que a adição de *plugins* podem torná-los IDEs.

2.3 Eclipse

Esta secção descreve sucintamente a origem e propósito do Eclipse. Para além disso, são descritos alguns conceitos importantes no Eclipse, tal como, algumas vantagens e desvantagens do mesmo.

Em 7 de novembro de 2001, um projeto de código aberto chamado Eclipse 1.0 foi lançado. Nessa altura o Eclipse foi descrito como "um ambiente de desenvolvimento integrado para qualquer coisa e nada em particular" (12). A descrição era propositadamente genérica porque a ambição da sua arquitetura não era ser um outro conjunto de ferramentas, mas sim uma *framework*. Especificamente, uma *framework* modular e escalável. Para alcançar tal ambição, a plataforma foi construída com pequenas unidades de funcionalidade, os *plugins*. É importante clarificar alguns conceitos, relativos aos *plugins*:

- **Plugin** – A melhor analogia em termos de *software* compara um *plugin* a um objeto duma linguagem orientada a objetos. Um *plugin* é um encapsulamento de comportamento e/ou dados que interagem com outros *plugins* formando um programa. Para interagir com outros *plugins* são necessárias extensões e pontos de extensão.
- **Manifesto** – Cada *plugin* é descrito por um manifesto. Esse manifesto é um ficheiro *XML* que descreve o *plugin*, as suas dependências, como pode ser utilizado, ou estendido. O manifesto é utilizado pela plataforma para carregar o *plugin* que descreve.
- **Extensões e pontos de extensão** – Um *plugin* permite que outro *plugin* estenda porções da sua funcionalidade ao declarar um ponto de extensão. A metáfora mais simples para descrever extensões e pontos de extensão são as tomadas elétricas. A tomada é o ponto de extensão, enquanto a ficha que se conecta, é a extensão. Tal como, as tomadas, os pontos de extensão tem uma ampla variedade de formas e tamanhos, e apenas extensões com certas características poderão se conectar a esse ponto de extensão. Essas características são declaradas pelo ponto de extensão num contrato. O contrato é tipicamente uma combinação de *XML* e interfaces Java, que as extensões devem seguir.

A plataforma Eclipse é estruturada em volta do conceito de *plugins*. Cada subsistema na plataforma é ele próprio estruturado como um conjunto de *plugins* que implementam alguma funcionalidade. Alguns *plugins* adicionam funcionalidades visíveis à plataforma usando o modelo de extensão, enquanto outros têm um caráter auxiliar ao fornecerem bibliotecas de classes que podem ser utilizadas para implementar extensões do sistema.

A arquitetura extensível do Eclipse foi uma das chaves para o grande crescimento do seu ecossistema. Com essa arquitetura, empresas ou indivíduos podiam desenvolver novos *plugins*, e lançar os mesmos como código aberto ou proprietário.

O Eclipse começou como uma plataforma e o Eclipse SDK foi o produto que serviu de prova de conceito. O Eclipse SDK (*Software Development Kit)* permite aos desenvolvedores utilizar o próprio para criar novas versões do Eclipse.

O SDK inclui a plataforma base e duas ferramentas principais que são utilizadas para o desenvolvimento de *plugins*. O JDT (*Java Development Tools*) implementa um ambiente de desenvolvimento Java com várias funcionalidades. O PDE (*Plug-in Development Environment*) adiciona ferramentas especializadas que agilizam o desenvolvimento de *plugins* e extensões. Estas ferramentas não só servem um propósito útil, como providenciam um ótimo exemplo de como novas ferramentas podem ser adicionadas à plataforma (figura 2) através do desenvolvimento de *plugins* que estendem o sistema.

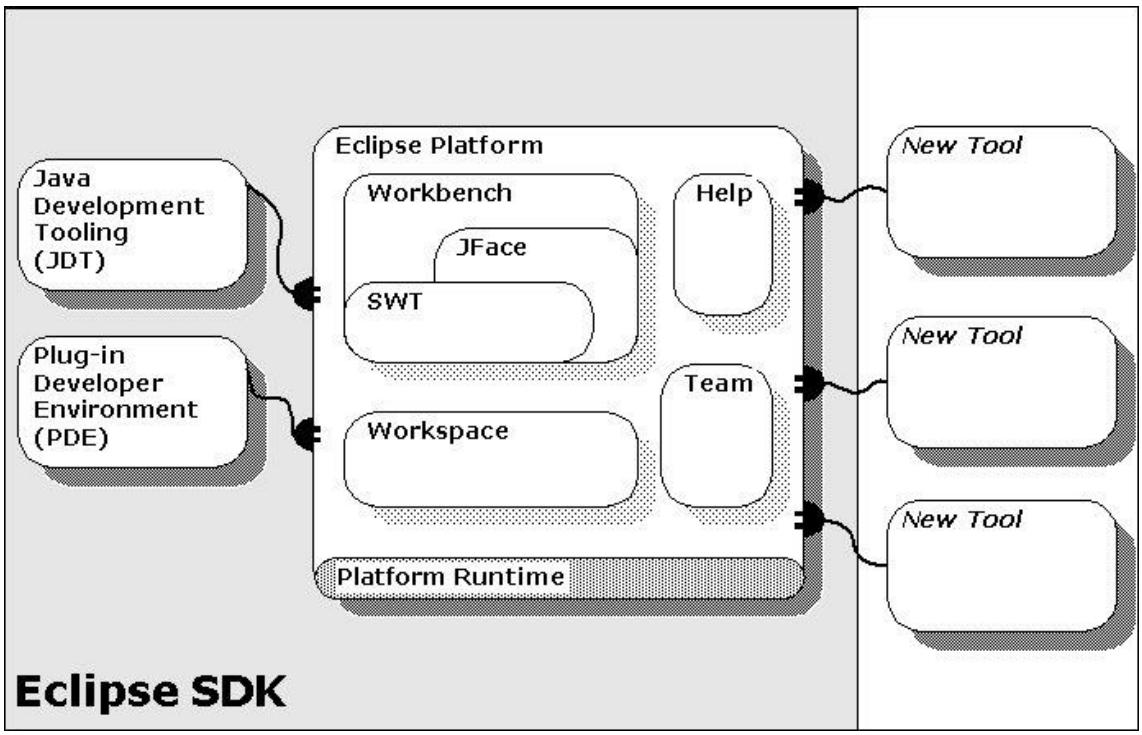


Figura 2 – Arquitetura da plataforma Eclipse

Embora a maioria dos utilizadores esteja satisfeita em usar o Eclipse como um ambiente de desenvolvimento integrado para Java, a plataforma dispõe dum grande número de *plugins* que permite o suporte de várias outras linguagens como C/C++, PHP, Prolog. Como descrito anteriormente, o Eclipse também inclui o *Plug-in Development Environment* (PDE), que é de interesse principalmente daqueles que desejam estender o Eclipse, visto que permite desenvolver ferramentas que se integram perfeitamente no ambiente do Eclipse. Como tudo no Eclipse é *plugin*, todos os desenvolvedores de ferramentas têm uma maneira uniforme de estender o Eclipse fornecendo um IDE unificado e consistente aos utilizadores.

Os utilizadores realçam positivamente diversos aspetos do Eclipse:

- ✓ Software livre.
- ✓ Portabilidade.
- ✓ Funcionalidades de refatoração (*refactoring*).
- ✓ A facilidade de atualizar *software* e instalar *plugins.*
- ✓ Disponibilidade das melhores ferramentas para o desenvolvimento de Java (linguagem de programação mais utilizada).
- ✓ Pequenas funcionalidades como integração de testes unitários (*JUnit*) ou utilização de perspetivas que disponibilizam uma fácil e rápida mudança de contexto.

De igual forma, também, são apontados alguns aspetos negativos:

- × Demasiado tempo para iniciar.
- × Documentação pouco concreta.
- × Apesar da grande comunidade, é bastante difícil esclarecer dúvidas através do fórum oficial[6].
- × As suas várias funcionalidades e conceitos tornam o Eclipse algo complexo.

[6] Fórum oficial do Eclipse URL: www.eclipse.org/forums

2.4 Microsoft Visual Studio

O Microsoft Visual Studio (figura 3) é um pacote de programas da Microsoft para desenvolvimento de *software* especialmente dedicado ao .NET *Framework* e às linguagens Visual Basic (VB), C, C++ e C# (C Sharp) (13). Também é um grande produto de desenvolvimento na área *web*, usando a plataforma do ASP.NET. O Visual Studio é comercializado, mas oferece algumas versões gratuitas, e foca principalmente duas áreas: desenvolvimento de aplicações para Windows e desenvolvimento *web*.

Enquanto o Eclipse é o ambiente de eleição para Java, o Visual Studio (VS) é considerado o ambiente de eleição para o desenvolvimento em C, C++ e C#.

O Visual Studio (*VS*) requer muito mais espaço para a sua instalação (cerca de 2.3 GB) que o Eclipse (cerca de 360 MB). Para além de mais espaço, o VS tem requisitos mínimos mais elevados que o Eclipse (14). O Eclipse é multiplataforma enquanto o VS é apenas suportado pelo Windows.

Em termos de usabilidade, ambos os IDEs disponibilizam vários menus, comandos e afins. No entanto, devido ao elevado número de opções, um inexperiente utilizador poderá perder-se. O Eclipse tem uma clara vantagem no campo da usabilidade que são as perspetivas. Uma perspetiva define um conjunto de janelas (*views*) e o seu *layout* no ambiente de trabalho do Eclipse. Um utilizador de Eclipse pode mudar de perspetiva reduzindo substancialmente a possível sobrecarga de opções do ambiente. O VS não oferece perspetivas por omissão.

Em termos de funcionalidades, tanto VS como Eclipse oferecem um leque bastante amplo e semelhante de funcionalidades.

Na extensibilidade o Eclipse tem uma vantagem notória perante o VS. Grande parte dessa vantagem se deve ao facto de que o Eclipse foi desenhado para ser um IDE extensível para todos os propósitos. O Eclipse facilita a sua extensão, e por isso, está disponível uma vasta gama de extensões para o mesmo. O VS é extensível, mas o desenvolvimento e inclusão de extensões na plataforma tende a ser complicado. Há alguns anos que a Microsoft tenta construir uma comunidade em volta das extensões para VS (14), mas até agora está aquém do sucesso que o Eclipse sempre teve.

Por fim, o Eclipse suporta um grande número de linguagens de programação, enquanto o VS é muito mais limitado na sua oferta.

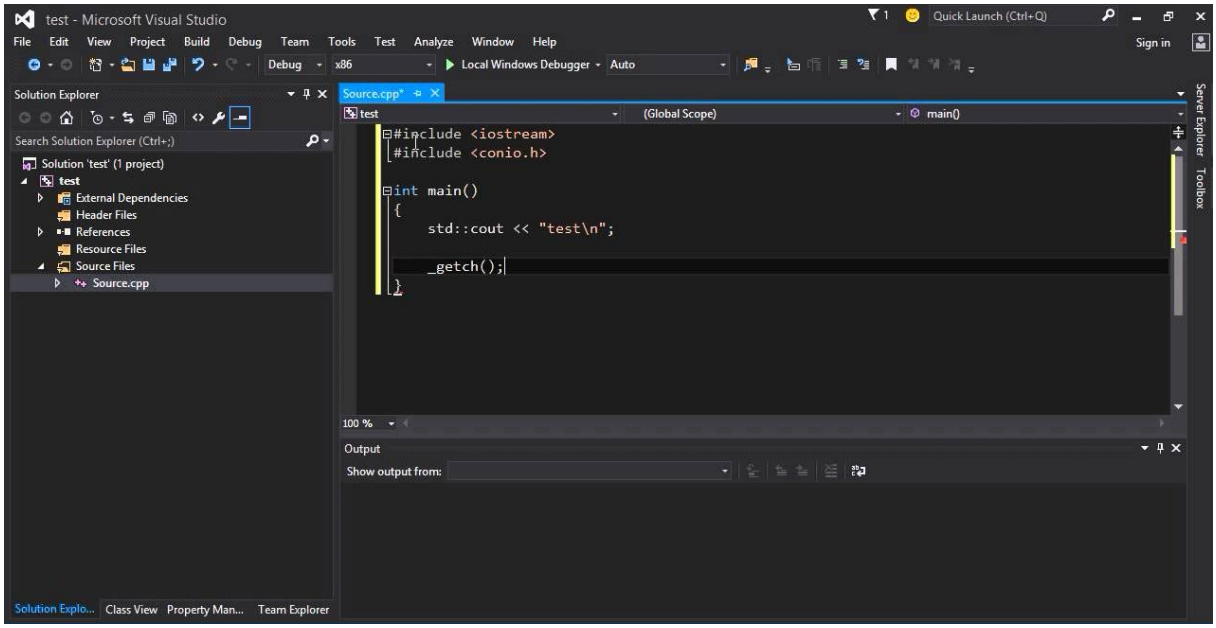


Figura 3 – Microsoft Visual Studio

2.5 NetBeans

O NetBeans (figura 4) é um ambiente de desenvolvimento integrado gratuito e de código aberto para desenvolvedores de *software* nas linguagens Java, C, C++, PHP, Groovy, Ruby, entre outras (15). O NetBeans é multiplataforma e oferece aos desenvolvedores ferramentas para criar aplicativos profissionais de *desktop*, empresariais, web e móveis multiplataformas.

Eclipse e NetBeans são bastante semelhantes sendo que ambos são plataformas livres e extensíveis, baseadas em Java, para o desenvolvimento de *software*.

Tanto o NetBeans como o Eclipse são extensíveis. Enquanto o Eclipse utiliza o conceito de *plugin* para a sua extensibilidade o NetBeans utiliza o conceito de módulo. Um módulo em NetBeans é um conjunto de classes Java que disponibilizam a uma aplicação uma funcionalidade específica. Apesar de *plugins* e módulos serem muito semelhantes, os *plugins* acarretam alguns conceitos como extensões e pontos de extensão que facilitam a sua interação com outros *plugins* agilizando o desenvolvimento de novas extensões à plataforma. A maior facilidade de criar extensões no Eclipse influencia a maior comunidade e leque de opções que o Eclipse possui perante o NetBeans.

O NetBeans utiliza Swing[7] na sua interface enquanto o Eclipse utiliza SWT[8]. Neste caso, o NetBeans leva a melhor já que a utilização de Swing em vez de SWT é mais vantajosa (16). Por exemplo, existem milhares de bibliotecas de terceiros disponíveis para Swing, o que significa que são fáceis de integrar nas aplicações da plataforma NetBeans.

Por último, o NetBeans não oferece perspetivas o que prejudica a sua usabilidade.

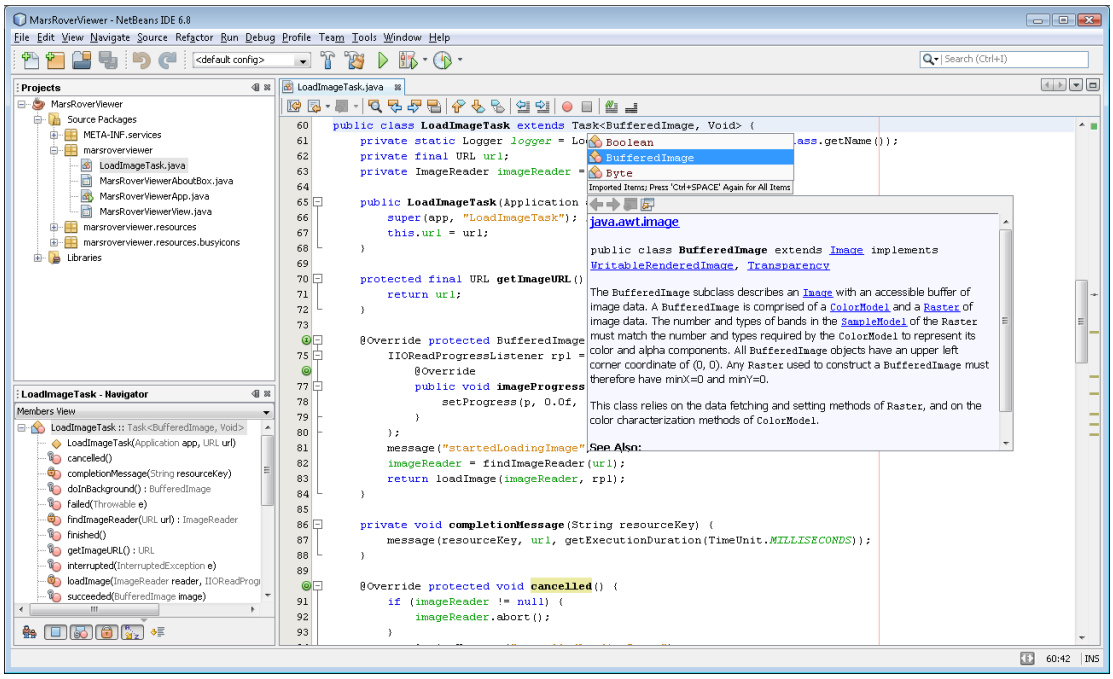


Figura 4 - NetBeans

[7] Mais informações sobre Swing em: en.wikipedia.org/wiki/Swing_(Java)
[8] Mais informações sobre SWT em: en.wikipedia.org/wiki/Standard_Widget_Toolkit

3 Prolog e seus IDEs

Neste capítulo, são apresentadas noções sobre Prolog e CxProlog, tal como exemplos de IDEs que suportam Prolog. Em alguns dos exemplos, é realizada uma breve comparação entre esses IDEs e os do CxIDE.

3.1 Prolog e CxProlog

O Prolog surgiu em 1972, sendo uma das primeiras linguagens de programação lógica, atualmente continua a ser uma das linguagens lógicas mais populares.

Programar em Prolog abre a mente para uma nova forma de olhar para a computação. Existe uma mudança de perspetiva que cada programador de Prolog experiência quando conhece a linguagem pela primeira vez (17). A abordagem do Prolog a um problema é, tendencialmente, mais de descrever factos e relações sobre um problema, e menos sobre prescrever uma sequência de passos dados pelo computador para resolver um problema (18). Para representar uma solução em Prolog escrevem-se predicados, que são sequências de cláusulas. Por sua vez, uma cláusula pode ser um fator ou uma regra. Um facto constitui um elemento de conhecimento verídico; uma cláusula pode ser vista como uma forma de extrair novos factos a partir dos existentes. Atualmente o Prolog é utilizado em domínios como: inteligência artificial, pesquisa na medicina e prospeção de dados (*data mining)* (19).

Historicamente, foram desenvolvidas várias implementações da linguagem de programação Prolog. No caso particular do CxProlog[9], trata-se duma extensão do Prolog que suporta a maioria da parte 1 do padrão (*standard*) ISO Prolog (20), e também o dialeto clássico de Edimburgo. Ao nível da linguagem, as principais extensões são: *units* (módulos), contextos, termos cíclicos, mecanismos imperativos e corrotinas. Ao nível da biblioteca, as principais extensões oferecidas são: interface para Java, interface para WxWidgets, suporte para Unicode, suporta para diversos tipos de estruturas de dados (tais como dicionários, vetores, pilhas, buffers) e ainda variáveis imperativas.

O CxProlog foi desenvolvido no NOVA LINCS[10] (NOVA Laboratório de Ciência da Computação e Informática) do Departamento de Informática, FCT, Universidade Nova de Lisboa sendo o seu criador o Professor A. Miguel Dias.

3.2 IDEs para Prolog

Muitos dos IDEs para Prolog são baseados em Eclipse, e apesar de partilharem um conjunto comum de funcionalidades, existem alguns IDEs que tentam aprofundar mais as suas funcionalidades tendo em conta as características do Prolog.

3.2.1 PDT - Prolog Development Tool

PDT[11] é um IDE para Prolog disponibilizado como um *plugin* para a plataforma Eclipse. Todas as funcionalidades do PDT estão implementadas para o SWI-Prolog [12], a maioria também está implementada para Logtalk[13]. Todas as ferramentas de desenvolvimento disponibilizadas pelo SWI-Prolog (traçador gráfico, depurador, perfilador (*profiler)*, ...) podem ser utilizadas no PDT (21).

O PDT dispõe de todas as funcionalidades descritas no capítulo 4. Existem algumas que são mais próprias do Prolog como a opção de executar golos (através da consola) de predicados Prolog

[9] Mais informações sobre o CxProlog em: ctp.di.fct.unl.pt/~amd/cxprolog
[10] Mais informações sobre NovaLincs em: nova-lincs.di.fct.unl.pt
[11] Mais informações sobre o PDT em: sewiki.iai.uni-bonn.de/research/pdt/docs/start
[12] Mais informações sobre SWI-Prolog em: www.swi-prolog.org/IDE.html
[13] Mais informaçõe sobre Logtalk em: logtalk.org

selecionados anteriormente ou a vista de contexto que permite visualizar: relações de chamada e propriedades de predicados.

O PDT é código aberto. Foi desenvolvido de forma a facilitar a integração de outros tipos de Prolog. É um dos IDEs recomendados para a utilização de SWI-Prolog. O PDT (figura 5) continua sobre uma constante monitorização e atualização por docentes da University of Bonn.

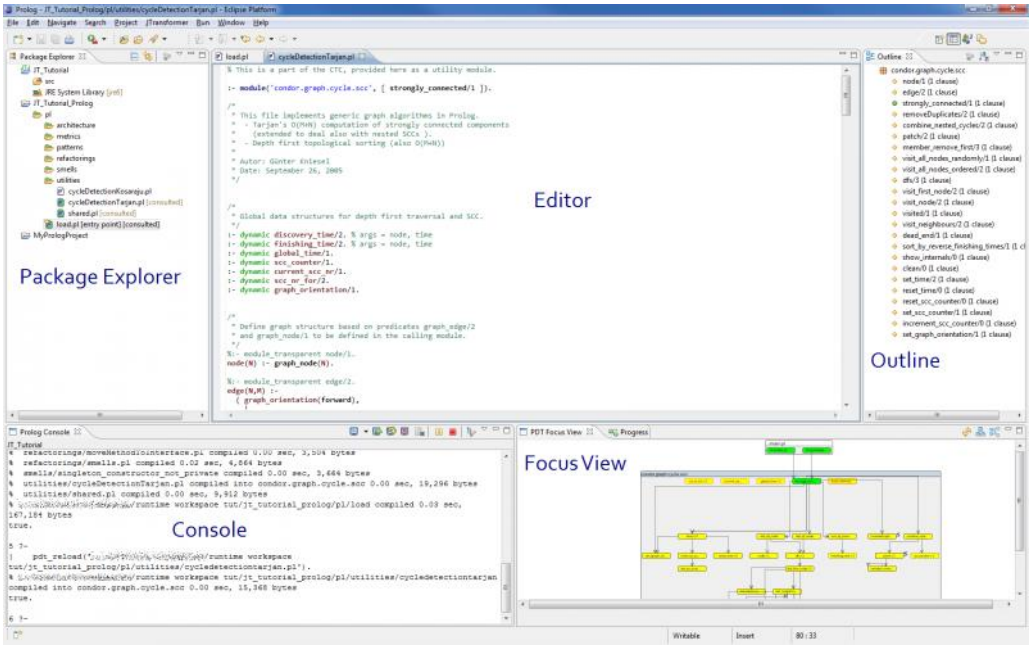


Figura 5 – Prolog Development Tool (PDT)

3.2.2 SPIDER - SICStus Prolog IDE

SPIDER é um IDE, baseado em Eclipse, para o SICStus Prolog. O SICStus é um sistema de desenvolvimento Prolog compatível com o padrão ISO (22).

Para além de implementar as funcionalidades descritas no capítulo 4, o SPIDER implementa[14]:

- Hierarquia de chamadas (clicar num predicado revela uma árvore dos seus chamadores e referências);
- Analisador de determinismo (os predicados são analisados para averiguar determinismo ou terminação);
- Analisador de modo (os predicados são analisados, e argumentos prováveis de saída são revelados quando pairando o cursor sobre um predicado em um golo);
- Prolog *toplevel* com o histórico de entradas.

O SPIDER é *software* proprietário sendo necessário adquirir uma licença para a sua utilização.

3.2.3 Ergo Studio/Prolog Studio para XSB Prolog

Tratam-se de duas variantes dum IDE para XSB Prolog, implementadas usando as linguagens Java e Prolog (23). A primeira variante é proprietária, constituindo na prática o IDE do sistema Ergo Suite da empresa Coherent Knowledge[15]. A segunda variante é um sistema de código livre. Ambas as variantes são da autoria do Doutor Miguel Calejo[16].

Para além das usuais funcionalidades dum IDE, o Studio tem algumas particularidades interessantes: certas perguntas podem ser feitas sob a forma duma tabela apresentada graficamente,

[14] Mais informações sobre o SicStus Prolog IDE em: sicstus.sics.se/spider
[15] Mais informações sobre a Coherent Knowledge em: coherentknowledge.com
[16] Mais informações sobre o PrologStudio em: interprolog.com/interprolog-studio

algumas respostas podem ser obtidas sob a forma dum grafo apresentado graficamente, e está disponível um grafo de chamadas onde é fácil de navegar usando o rato.

Neste IDE, existem algumas semelhanças com os objetivos da corrente dissertação, sendo por isso importante fazer uma boa análise comparativa. Para começar, vejamos os aspetos comuns entre este IDE e o CxIDE :

- O IDE está programado parcialmente em Prolog e parcialmente em Java.
- Recorre a elevada integração entre Prolog e Java, usando mecanismos de interoperabilidade.

Vejamos agora alguns pares de diferenças:

1 **XSB Prolog** - A solução depende do XSB Prolog e do mecanismo de interoperabilidade InterProlog. Para funcionar, estes dois elementos são requeridos.

1 **CxProlog** - A solução destina-se a funcionar no CxProlog, usando os mecanismos de interoperabilidade disponíveis no CxProlog.

2 **XSBProlog** - A solução é autónoma e recorre a diversas bibliotecas disponíveis para Java e ao Swing (o GUI para Java).

2 **CxProlog** - A solução destina-se a funcionar no contexto do Eclipse, aproveitando o mais possível do que o Eclipse oferece, e tentando ter as melhores características dum *plugin* típico do Eclipse.

Finalmente duas questões que são muito relevantes no projeto de dissertação proposto, mas que a documentação do Ergo Studio/Prolog Studio não refere:

- ✓ O IDE para CxProlog destina-se a ser facilmente configurável e adaptável pelo utilizador usando código Prolog.
- ✓ O IDE também constituirá um ambiente de execução, e durante a execução existirá a liberdade de instalar novos menus, diálogos, etc.

3.2.4 Plugin CxProlog para CodeBlocks

Na referência (24) é descrito um outro IDE para CxProlog que têm objetivos muito semelhantes ao deste trabalho. Inclusivamente, procura-se lá que o IDE seja extensível através da linguagem Prolog, esse trabalho também teve o mesmo orientador do presente trabalho.

Sendo verdade que os objetivos gerais são aproximadamente os mesmos, a verdade é que as soluções tecnológicas encontradas assim como o nível de sucesso são completamente díspares. Esse trabalho correspondia a um *plugin* para o IDE CodeBlocks[17] escrito em C/C++. Nessa altura o CodeBlocks estava muito pouco desenvolvido e apesar de teoricamente ele suportar uma arquitetura de *plugins* aquilo que ele oferecia era demasiado orientado para o único *plugin* existente na altura, um *plugin* para o C++. Portanto, o *plugin* para CxProlog foi obrigado a programar quase tudo de raiz, inclusivamente a janela do projeto e a consola. Em consequência as funcionalidades não podiam ser muito ambiciosas mesmo ao nível da edição de código. Nessa altura também não foram usadas as corrotinas do CxProlog o que tornou a implementação da interação da consola pouco realista devendo apenas ser considerada um protótipo modesto a aguardar melhor solução. Em todo o caso, não se pense que era um mau projeto. Pelo contrário era um trabalho muito válido considerando as limitações tecnológicas envolventes.

[17] Mais informações sobre CodeBlocks em: www.codeblocks.org

Portanto, resumidamente, há três aspetos muito diferentes entre o IDE do CodeBlocks e o IDE deste trabalho:

- O *plugin* para CodeBlocks foi escrito C/C++, enquanto o CxIDE está escrito maioritariamente em Java utilizando interoperabilidade não trivial.

- O CodeBlocks é uma plataforma bastante incipiente com poucas funcionalidades acessíveis a novos *plugins*, dessa forma o *plugin* para o CodeBlocks esteve muito condicionado pelos poucos recursos disponíveis. Por outro lado, o CxIDE foi desenvolvido no contexto do Eclipse, onde existe uma imensidão de recursos para explorar.

- O *plugin* para CodeBlocks apresenta uma consola que se baseia numa implementação bastante frágil e pouco realista do *toplevel* do CxProlog. A consola interna do CxIDE, apresenta uma robusta implementação baseando-se no *toplevel* original do CxProlog a correr numa corrotina própria.

4 Funcionalidades e ferramentas dos IDEs

Os IDEs, ao longo de vários anos evoluíram, o número de funcionalidades e ferramentas que disponibilizavam aumentava, tal como o número de linguagens de programação suportadas. Cada vez mais programadores eram cativados a utilizar IDEs.

Um estudo realizado em 2013 (1), colocou a questão "Que tipo de ferramentas utiliza no desenvolvimento de *software*?", as respostas são reveladas na figura 6.

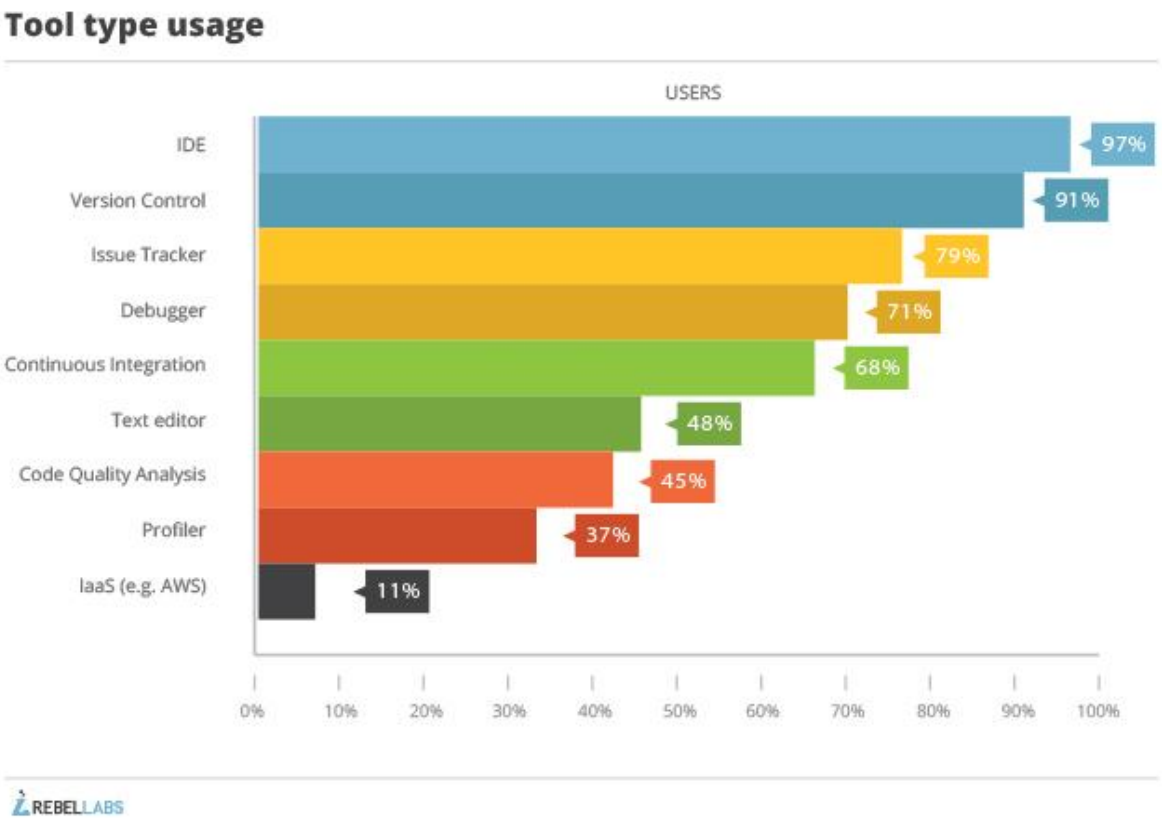


Figura 6 - Estudo de 2013 sobre as ferramentas utilizadas no desenvolvimento de software

O gráfico revela que o uso de IDEs no desenvolvimento de *software* é atualmente algo quase indispensável.

Mas quais as razões que levaram os desenvolvedores de *software* a valorizar tanto os IDEs? Existem várias razões, sendo a mais importante o aumento da produtividade. O que possibilita este aumento de produtividade são as funcionalidades e ferramentas que os IDEs disponibilizam através de uma interface comum. O remanescente desta secção focará algumas funcionalidades e ferramentas que a maiorias dos IDEs disponibilizam aos seus utilizadores, tal como os seus benefícios.

4.1 Funcionalidades

Nesta secção são enumeradas as funcionalidades mais comuns dos IDEs atuais. É realizada uma breve descrição dessas funcionalidades, realçando os seus benefícios e revelando alguns exemplos visuais. Os exemplos desta secção são do IDE para Java do Eclipse, pois este é um dos ambientes mais reconhecidos pelos programadores.

4.1.1 Realce de sintaxe

Oferece formatação específica para certas categorias de termos (figura 7). A mudança da formatação envolve alterações da fonte tipográfica, e principalmente, da coloração do texto. Aumenta a legibilidade do código e permite ao utilizador rapidamente localizar símbolos específicos da linguagem de programação em uso.

```
public void a_method(){
    //this is a comment
    System.out.println("A string");
}
```

Figura 7 – Realce de sintaxe dum excerto de código Java no Eclipse

4.1.2 Realce de erros

Semelhante ao realce de sintaxe, mas apenas altera a formatação dos erros no código (figura 8). Possibilita ao utilizador uma rápida deteção de erros sem muitas vezes ser necessário compilar o código.

```
publc void a_method(){
    //this is a comment
    System.out.rintln("A string");
}
```

Figura 8 – Realce de erros dum excerto de código Java no Eclipse

4.1.3 Validação sintática

Verifica se o código escrito está de acordo com as regras de sintaxe da linguagem de programação em uso. Caso não esteja poderá ser utilizado o realce de erros para notificar o utilizador. Erros de sintaxe são rapidamente detetados possibilitando uma pronta correção dos mesmos por parte do utilizador. Novos utilizadores da linguagem de programação beneficiam igualmente desta funcionalidade pois rapidamente são alertados para os seus erros de sintaxe o que lhes permite uma mais rápida compreensão da mesma.

4.1.4 Completação automática de código

Funcionalidade que oferece sugestões de completação de código (figura 9) reduzindo os erros de digitação e a memorização que o utilizador necessita ter sobre os elementos da linguagem de programação. Normalmente esta funcionalidade apresenta as suas sugestões através de *popups* aquando da edição do código, sendo as sugestões baseadas no atual contexto. Caso o utilizador opte por uma sugestão, o IDE incluirá essa sugestão no código em edição.

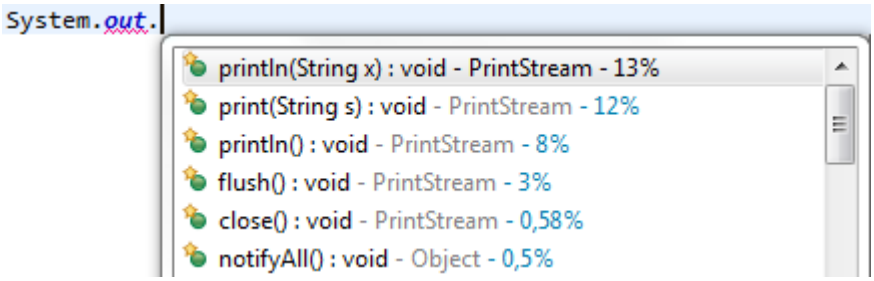


Figura 9 – Completação de código do IDE Java do Eclipse

A completação inteligente de código é uma das mais aclamadas funcionalidade dos IDEs e das que mais influência apresenta no aumento da produtividade do utilizador (25).

4.1.5 Navegação no código

Permite navegar rapidamente pelo código. Exemplos: abrir declaração, pesquisa de referências/definições.

4.1.6 Editor de código fonte

Um editor de texto que inclui funcionalidades (algumas acima referidas) que facilitam a edição do código fonte em causa.

4.1.7 Gestão de recursos

Durante o desenvolvimento de aplicações, as linguagens de programação normalmente confiam que certos recursos, como bibliotecas ou *header files*, estejam numa localização específica. Os IDEs devem conseguir gerir estes recursos e alertar para o caso em que os recursos necessários estejam em falta. Dessa forma podem ser detetados erros na fase de desenvolvimento que de outra forma só seriam detetados aquando da compilação ou construção.

4.1.8 Refatoração

É o processo de modificar um sistema de *software* para melhorar a estrutura interna do código sem alterar o seu comportamento externo (26). As modificações são realizadas automaticamente, não sendo necessário ao utilizador percorrer todas as dependências afetadas pela modificação de forma a assegurar que o comportamento do programa não foi alterado. É necessário que um IDE tenha um profundo conhecimento duma linguagem para conseguir oferecer várias opções de refatoração sobre essa mesma linguagem. Existem vários tipos de refatoração (27), mas alguns tipos só são aplicáveis a certas linguagens de programação (figura 10).

Exemplos de refatorações:

- Renomeação – modificar o nome de por exemplo uma variável ou método para um novo nome que revela melhor o seu propósito.
- Generalizar tipo - criar tipos mais gerais, para permitir mais partilha de código.
- Extrair método – para transformar parte de um grande método em um novo método. Partir o código em pedaços menores torna-o mais compreensível. Esta refatoração também é aplicável a funções.

A refatoração beneficia a manutenção e extensibilidade da aplicação (28).

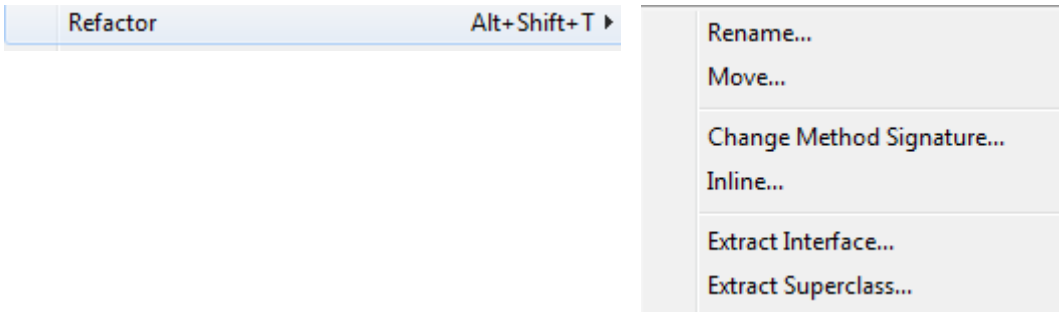


Figura 10 – Algumas opções de refatoração oferecidas pelo Eclipse, relativas à linguagem Java

4.1.9 Vistas estruturadas

Tratam-se de representações visuais que permitem ao utilizador visualizar informações de forma simples e compacta (figura 11).

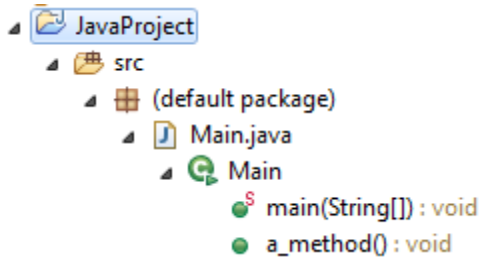


Figura 11 – Package Explorer, uma das vistas estruturadas do Eclipse no ambiente Java

Importante realçar que muitas das funcionalidades acima referidas poderiam ser realizadas manualmente (e de facto no passado eram) pelo programador. No entanto, seriam tarefas penosas que levariam muito tempo, assim ao fornecerem estas funcionalidades os IDEs permitem aos seus utilizadores dedicarem o seu tempo ao que realmente importa, a programação, aumentando a produtividade dos mesmos.

4.2 Ferramentas

Nesta secção são enumeradas e descritas algumas das ferramentas que os IDEs modernos integram. São revelados exemplos de cada uma das ferramentas, sendo esses exemplos provenientes dos IDE para Java e OCaml[18] suportados em Eclipse.

4.2.1 Compilador

Um compilador é um programa tradutor com as seguintes características:

- Converte programas duma linguagem de programação de alto nível para programas equivalentes escritos numa linguagem de mais baixo nível.
- A linguagem de mais baixo nível é destinada a uma máquina real ou virtual, permitindo que o programa possa ser executado diretamente na plataforma computacional.
- O programa executável, gerado pelo compilador, pode ser corrido as vezes que se quiser, sem ser preciso voltar a usar o compilador.

Exemplo: `javac` (compilador de Java)

4.2.2 Gerador de documentação

Um gerador de documentação é uma ferramenta de programação que gera documentação de *software* destinada a programadores, a utilizadores, ou a ambos. A documentação é gerada a partir de um conjunto de ficheiro de código fonte especialmente comentados, e em alguns casos, ficheiros binários.

Exemplo: `javadoc` (gerador de documentação de Java)

4.2.3 Profiler

O *profiler* analisa dinamicamente um programa realizando várias medições. Medições como: a memória utilizada, complexidade temporal do programa, a utilização de instruções particulares, ou a frequência e duração de chamadas de funções. Geralmente, as informações extraídas por o *profile*r são utilizadas para otimizar o programa.

---

[18] Mais informações sobre o OcaIDE em: www.algo-prog.info/ocaide

Exemplo: `ocamlcp` (profiler do OCaml)

4.2.4 Interpretador

Um interpretador é um programa "executor" com as seguintes características:

- Executa diretamente o programa fonte.
- Para correr o programa novamente é necessário voltar a usar o interpretador

Exemplo: `ocaml` (interpretador de OCaml)

4.2.5 Ligador

Depois de compilado o código fonte, para se obter o programa executável é preciso usar um programa ligador que junta os diversos ficheiros objeto num único programa executável. Na altura de ligar um programa é preciso também indicar quais são as bibliotecas (arquivos de ficheiros, objetos predefinidos) que interessa juntar ao programa.

Exemplo: em Linux, o ligador chama-se `ld` e pode ser invocado diretamente pelo utilizador.

4.2.6 Depurador

É um programa de computador usado para testar outros programas e fazer sua depuração, que consiste em encontrar os erros do programa.

Exemplo: `ocamldebug` (depurador de OCaml)

4.2.7 Construtor *(builder)*

Agrega vários tipos de ficheiros e configurações produzindo uma aplicação.

Existem outras ferramentas como geradores de *parsers* (ex: `ocamlyacc`) e geradores de reconhecedores de expressões regulares (ex: `ocamllex`) que os IDEs podem disponibilizar aos seus utilizadores.

4.3 Vantagens e Desvantagens dos IDEs

Esta secção enumera e descreve algumas importantes vantagens e desvantagens dos IDEs.

4.3.1 Vantagens

Menor tempo e esforço – A finalidade dos IDEs é tornarem o desenvolvimento mais rápido e fácil. As suas ferramentas e funcionalidades tentam cobrir o máximo de tarefas possíveis, para que o utilizador se possa focar na parte mais importante, a programação.

Impor padrões de projeto ou empresa – Ao trabalhar num mesmo IDE, um grupo de programadores seguirá certos padrões. Esse padrões podem ser ainda mais reforçados se o IDE oferecer modelos (*templates*) predefinidos, ou se bibliotecas de código forem partilhadas entre diferentes programadores do mesmo projeto.

Gestão de projeto – A maioria dos IDEs oferece ferramentas de documentação que facilitam entrada ou geração de comentários em diferentes partes do projeto. As vistas estruturadas também são um grande auxílio na gestão de recursos, pois apresentam de forma compacta os recursos utilizados, mesmo que estes estejam em diferentes diretorias.

4.3.2 Desvantagens

Curva de aprendizagem – Um IDE pode ser uma ferramenta complicada. É necessário algum tempo e paciência para o aprender a usar bem e a maximizar os seus benefícios.

Um IDE sofisticado pode não ser uma boa ferramenta para programadores iniciantes – De facto, os IDEs podem facilitar a aprendizagem da linguagem que suportam. No entanto, a adição da aprendizagem do IDE à da linguagem que suporta pode causar problemas a iniciantes. Os IDEs mais

sofisticados podem ocultar alguns aspetos básicos das linguagens, o que pode gerar problemas na aprendizagem dessa linguagem. É importante que interessados em IDEs realizem uma escolha ponderada sobre quais os IDEs que mais os beneficiam e se adequam aos seus conhecimentos.

Não corrigem más práticas de programação ou desenho – O programador necessita de ser proficiente e meticuloso. Muito provavelmente o IDE não conseguirá eliminar problemas de legibilidade, bom uso dos recursos e desempenho. O IDE não tem todas as responsabilidades na produção de *software* de qualidade, sendo que a maioria das responsabilidades são do programador.

É importante realçar que apesar de os IDEs aumentarem a produtividade a vários níveis no desenvolvimento de aplicações, a complexidade do ambiente pode ter uma influência negativa principalmente ao nível da depuração e aprendizagem (25). E nem sempre um IDE com mais funcionalidades e ferramentas é melhor do que um com menos, é necessário ter cuidado com a complexidade que adicionais funcionalidades e ferramentas acrescentarão ao ambiente.

5 Estender o Eclipse

Qualquer funcionalidade do Eclipse é disponibilizada através de *plugins*. Por exemplo, as ferramentas de desenvolvimento Java são acessíveis como um conjunto de *plugins*. Assim, as funcionalidades de desenvolvimento Java apenas estão disponíveis se os *plugins* que as definem estiverem presentes na instalação do Eclipse.

A funcionalidade do Eclipse é fortemente baseada nos conceitos de extensão e ponto de extensão (*secção 2.3*). Por exemplo o IDE para Java do Eclipse disponibiliza um ponto de extensão para adicionar novos modelos (*templates*) para o editor Java.

É possível contribuir para uma funcionalidade existente na plataforma através da adição de *plugins*, por exemplo novas entradas de menu, novas entradas na barra de ferramentas ou disponibilizar uma funcionalidade completamente nova. De forma mais radical, é também possível criar novos ambientes de programação.

5.1 Pré-requisitos

Os pré-requisitos mínimos para desenvolver uma extensão à plataforma Eclipse são os seguintes:

- ✓ Instalação do Eclipse SDK (contém o Plug-in Development Environment já descrito)[19].
- ✓ Instalação do JRE (Java Runtime Environment)[20].
- ✓ Conhecimentos no desenvolvimento em Java.
- ✓ Conhecimentos de desenvolvimento na plataforma Eclipse.

No entanto, é importante realçar que consoante a complexidade do projeto planeado serão necessários outros tipos e níveis de conhecimentos. Por exemplo, para desenvolver um IDE no Eclipse para uma linguagem de programação será necessário deter conhecimentos sobre as interfaces que as ferramentas dessa linguagem disponibilizam para integrar as mesmas com sucesso.

5.2 Simples exemplo de adição de nova funcionalidade ao Eclipse

Será demonstrado seguidamente como adicionar uma nova funcionalidade à plataforma Eclipse. Neste exemplo serão revelados os passos necessário para criar um projeto de desenvolvimento de *plugins* e como adicionar um novo botão e menu, à barra de ferramentas e barra de menus respetivamente. Esse botão deterá uma simples funcionalidade: revelar uma janela de diálogo com a seguinte sequência de carateres "Hello, Eclipse world".

Primeiro, no Eclipse SDK criar um novo projeto *plugin* (*plug-in project*) com o nome *com.ramos.plugin.first* (ou outro se desejado) através de:

File → New → Project... → Plug-in Development → Plug-in Project

Ao utilizador serão apresentados vários campos de um *wizard* que poderão ser preenchidos como na figura 12.

[19] Descarregar Eclipse SDK em: download.eclipse.org/eclipse/downloads
[20] Descarregar JRE em: goo.gl/NNGKVVcontent_copy

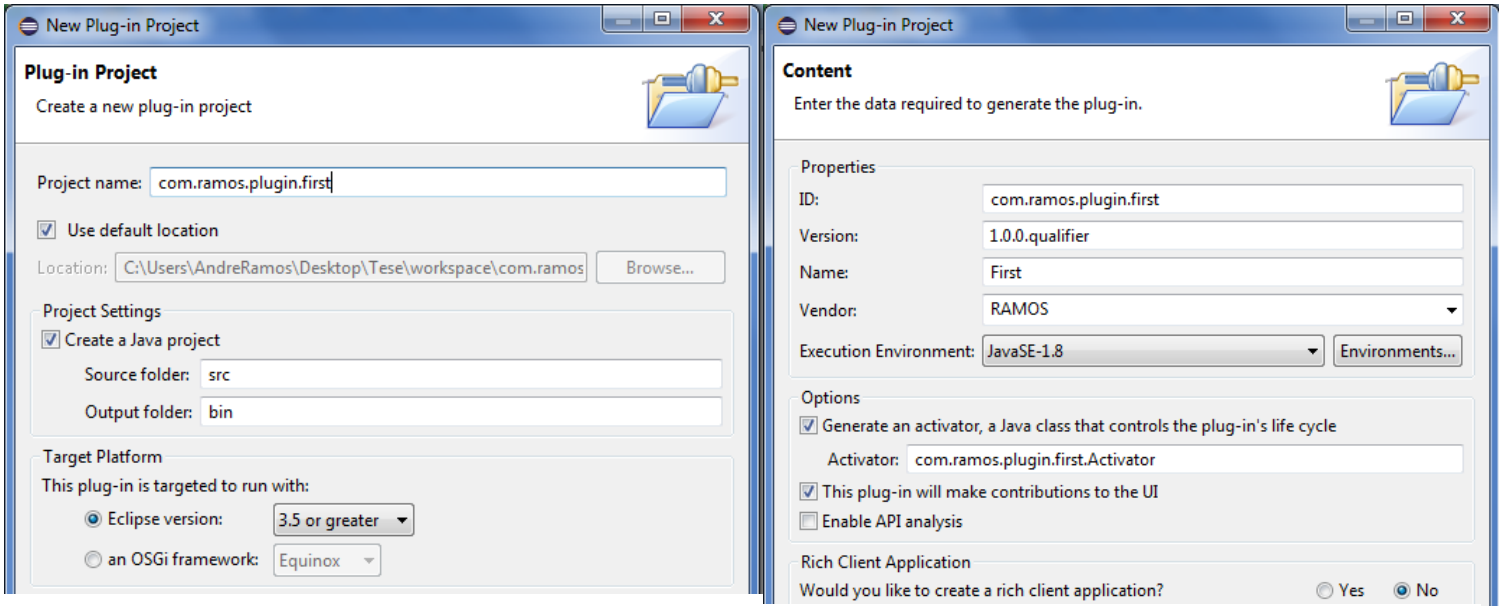


Figura 12 – Fase inicial do wizard para criação dum projeto plugin

No seguinte *wizard* (figura 13) existem vários modelos padrão disponíveis que oferecem uma implementação base facilitando a fase inicial dum projeto. Os modelos oferecem várias implementações base, desde um *plugin* com editor a *plugin* com um simples novo comando. Neste caso, escolheremos a opção "Hello, World Command" que fornece toda a implementação que define a adição de um novo botão à barra de ferramentas e de um novo menu com uma única opção, tanto o botão e opção invocam a mesma ação que é revelar uma janela de diálogo.

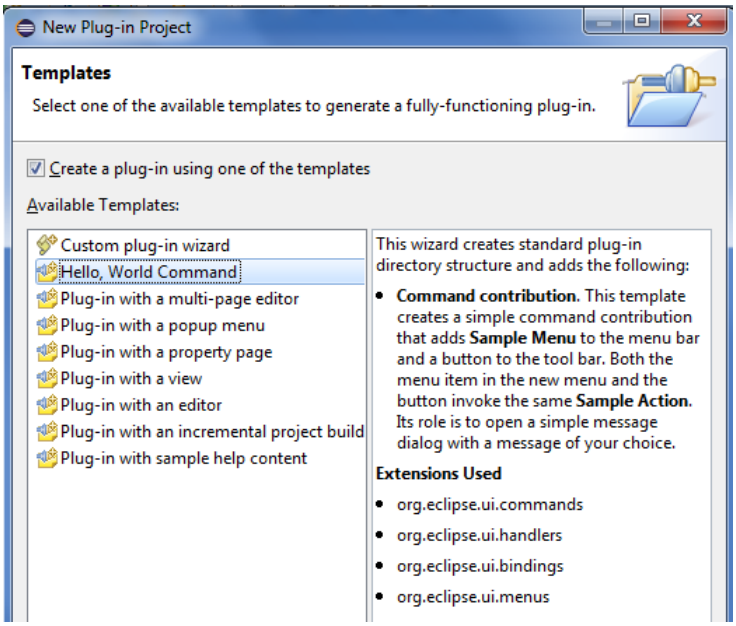


Figura 13 – Fase de seleção dum modelo plugin no wizard de criação dum projeto *plugin*

Após a fase representada na figura 13, o utilizador poderá clicar em **Finish** para criar o seu projeto *plugin*. Poderá então surgir uma notificação para mudar a perspetiva atual do Eclipse para o PDE, e nesse caso, o utilizador deverá aceitar a mudança. Toda a implementação necessária para o funcionamento do modelo escolhido já estará concluída e pronta a executar. Para executar um projeto *plugin*, basta selecionar com o botão da direita:

Run as → Eclipse Application

O utilizador terá agora um projeto *plugin* com a estrutura revelada na figura 14. O projeto utiliza quatro extensões para implementar as suas funcionalidades:

- **org.eclipse.ui.commands** - para declarar comandos e categorias de comandos. Um comando é uma representação abstrata de algum comportamento semântico. No exemplo em causa, esta extensão é utilizada para o botão na barra de ferramentas e a opção do novo menu.
- **org.eclipse.ui.handlers** - um *handler* é o responsável por o comportamento dum comando em um determinado ponto do tempo. No exemplo em causa, o *handler* associa o botão e a opção de menu ao comportamento definido na classe Java **SampleHandler**.
- **org.eclipse.ui.bindings** - utilizado para declarar ligações (*bindings*) ou esquemas. Esquemas são conjuntos de ligações. Uma ligação é um mapeamento entre um certo grupo de condições, algum tipo de *input* e um comando acionado. No exemplo em causa, a ligação é utilizada para invocar através do teclado (*crtl + 6*) a mesma ação que o botão e opção de menu executam.
- **org.eclipse.ui.menus** - possibilita a adição de comandos à barra de ferramentas, barra de menus, menus e a outros elementos da interface. No exemplo em causa, esta extensão é responsável por adicionar o novo botão, menu e opção à barra de ferramentas, barra de menus e menu respetivamente.

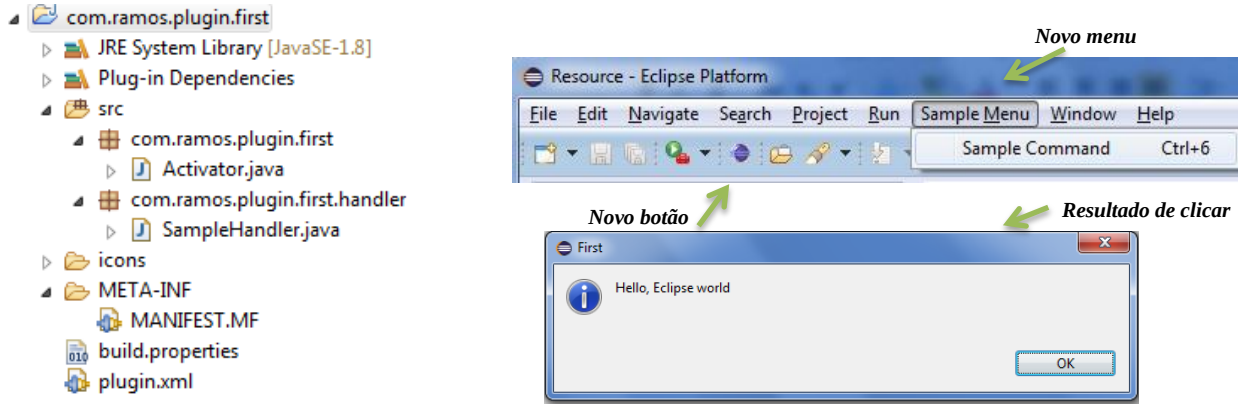


Figura 14 – Estrutura do projeto plugin e resultado do mesmo

Assim o que torna simples a implementação desta nova funcionalidade no Eclipse é o facto de o Eclipse dispor de várias extensões prontas a serem utilizadas. Estas extensões são como peças de um puzzle que podem ser utilizadas para construir uma nova e maior peça (funcionalidade) dependente das anteriores. A figura 15 revela esquematicamente como as quatro extensões foram utilizadas para implementar a nova funcionalidade.

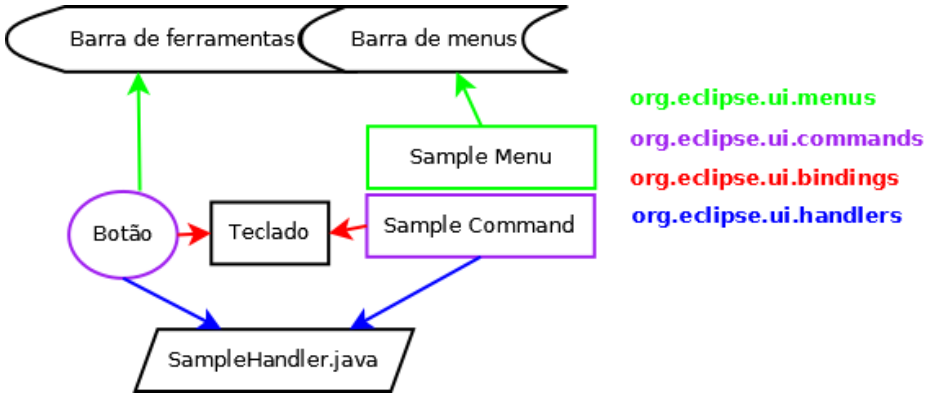


Figura 15 – Esquema de interação entre vários componentes do *plugin* e os pontos de extensão utilizados

O desenvolvedor pode através de *wizards* modificar alguns atributos das extensões utilizadas (ex: imagem (ícone) ou localização do botão). No exemplo em causa, a maior liberdade é sobre o

comportamento dos comandos, sendo que o desenvolvedor pode modificar a classe *Java* **SampleHandler** (abaixo) para alterar esse comportamento. Por exemplo, seria possível enviar o ficheiro em edição no editor para um compilador e nesse caso a nova funcionalidade seria um atalho para compilar o ficheiro em edição.

```
public class SampleHandler extends AbstractHandler {
    public SampleHandler() {
    }

    /**
     * Aquando de uma interação com os novos elementos de interface
     * criar uma janela de diálogo com o título first e a mensagem
     * Hello, Eclipse World"
     */
    public Object execute(ExecutionEvent event) throws ExecutionException {
        IWorkbenchWindow window = HandlerUtil.getActiveWorkbenchWindowChecked(event);
        MessageDialog.openInformation(
                window.getShell(),
                "First",
                "Hello, Eclipse world");
        return null;
    }
}
```

6 Planeamento do CxIDE

Neste capítulo são descritas decisões iniciais que foram tomadas antes da implementação do projeto. Tais decisões influenciaram diretamente a implementação do projeto, sendo importante esclarecer o utilizador sobre as mesmas antes da descrição de todo o processo de implementação.

Inicialmente, existiam muitas dúvidas que necessitavam ser prontamente esclarecidas antes de iniciar a implementação do projeto...

6.1 Um IDE completamente configurado através do CxProlog?

Para se obter a configurabilidade de forma elegante, a primeira ideia foi criar um IDE em tempo de arranque desprovido de menus e de outros elementos gráficos. Basicamente, seria um ficheiro de configuração escrito em CxProlog que instalaria todos os elementos iniciais, isto é, o *layout* do Eclipse só seria composto pelos elementos (menus, janelas, etc.) definidos no ficheiro de configuração. Contudo, esta ideia não se revelou prática de realizar, pelos seguintes motivos:

- Seria necessário remover ou ocultar os elementos que o *layout padrão* do Eclipse (menu "File", vista "Project Explorer") oferece o que poderia comprometer o princípio da imitação.
- Não seriam aproveitadas muitas das facilidades de extensibilidade oferecidas pelo Eclipse.
- O projeto final seria mais um IDE *standalone* do que um *plugin* para Eclipse, o que iria contra os objetivos propostos.

Assim, aceitou-se que fosse o Java a criar e instalar a maioria dos elementos iniciais, os quais podem, contudo, ser alvo de alguma configuração em CxProlog.

6.2 Como será a comunicação entre o CxProlog e o Java?

Como foi descrito anteriormente (secção 2.3), o Eclipse é implementado em Java. No entanto, o projeto proposto pretende que o CxProlog tenha o máximo controlo possível sobre o *plugin* CxIDE. Assim, é necessário que Java e CxProlog comuniquem entre si.

Projetos de *software* que envolvam mais do que uma linguagem de programação requerem que essas mesmas linguagens suportem mecanismos que permitam a comunicação entre ambas. São os chamados mecanismos de interoperabilidade.

Para existir interoperabilidade entre duas linguagens são necessárias pelo menos duas coisas:

1. Cada linguagem tem de permitir a chamada de funções externas escritas na outra linguagem.
2. Tem de haver possibilidade de troca de dados entre as duas linguagens usando formatos convencionados.

Uma forma de alcançar a interoperabilidade é através duma FFI (*Foreign Function Interface*). A FFI é uma biblioteca que fornece uma determinada API e que permite a um programa escrito numa linguagem de programação chamar rotinas ou usar serviços escritos em outra linguagem. Uma conhecida FFI é a JNI(*Java Native Interface*) que estabelece interoperabilidade entre C e Java. Outra forma de alcançar interoperabilidade é através da utilização de *sockets*. *Sockets* são mecanismos oferecidos pelos sistemas operativos para que dois processos possam comunicar entre si.

Assim para desenvolver o CxIDE será necessária a utilização de interoperabilidade para que Java e CxProlog possam comunicar entre si.

O Java é utilizado para implementar as partes gráficas e tudo o está diretamente ligado à arquitetura do Eclipse. Enquanto o CxProlog é responsável pelo controlo geral do IDE e pela utilização de tudo o que foi disponibilizado pelo lado do Java.

Foram ponderadas duas hipóteses para possibilitar Java e CxProlog comunicarem entre si: *sockets* ou bibliotecas dinâmicas.

A utilização de *sockets* na implementação de IDEs, quando comparada com bibliotecas dinâmicas, têm as seguintes vantagens (indicadas por ✓) e desvantagens (indicadas por ×):

✓ Possibilidade de executar golos numa máquina externa mais poderosa, enquanto o IDE corre numa máquina mais lenta. Por exemplo, um IDE com uma consola implementada com *sockets* teria a possibilidade de conectar a sua consola a um processo de Prolog a correr em outra máquina.

✓ Maior facilidade para integrar outras implementações da mesma linguagem suportada pelo IDE. Por exemplo, uma consola implementada com *sockets* teria a possibilidade de intercambiar o processo Prolog utilizado.

✓ Se a comunicação entre IDE e linguagem suportada tem uma granularidade grosseira a implementação é facilitada.

× Se a comunicação entre IDE e linguagem suportada tem uma granularidade fina a implementação é dificultada, sendo necessária a estipulação dum rígido protocolo de comunicação.

× Interação entre IDE e a linguagem que suporta está limitada pelo protocolo de comunicação existente.

× Maior dificuldade na instalação. É necessário instalar o IDE e o ambiente de programação da linguagem que suporta separadamente. Sendo depois necessário que o utilizador configure o IDE para que o mesmo encontre os ficheiros necessários.

As bibliotecas "dinâmicas" também conhecidas como bibliotecas *shared* no caso do Linux são bibliotecas carregadas pelos programas em tempo de execução, ou seja dinamicamente.

O CxProlog possibilita a geração de uma biblioteca dinâmica através do comando **make lib**, sendo gerada a biblioteca **libcxprolog.so** em Linux.

A utilização duma biblioteca dinâmica na implementação dum IDE, quando comparada à solução com *sockets*, tem as seguintes vantagens e desvantagens, que advêm essencialmente do facto da biblioteca estar disponível localmente:

✓ Mais desempenho e mais eficiência.

✓ Maior nível de integração.

✓ Maior facilidade de instalação. Não é necessária a instalação do ambiente de programação da linguagem suportada.

✓ Se a comunicação entre IDE e linguagem suportada tem uma granularidade fina a implementação é facilitada.

× Maior dificuldade na integração de outras implementações da mesma linguagem suportada pelo IDE. O IDE só conseguirá utilizar a implementação da linguagem disponibilizada pela biblioteca dinâmica.

× Impossibilidade de executar a linguagem suportada numa máquina diferente.

× Dificuldade em utilizar diferentes versões.

Em termos de espaço de memória não há diferenças entre uma implementação com *sockets* ou com biblioteca dinâmica. O que acontece é que no caso dos *sockets* esse espaço pode ser distribuído entre mais computadores.

De realçar, que com um protocolo de comunicação rigoroso qualquer granularidade de comunicação poderia ser realizada através de *sockets*; o grande problema neste caso é a dificuldade em implementar tal protocolo.

Uma das grandes ambições deste projeto é dar o máximo de controlo ao CxProlog sobre o IDE. Por isso, é necessária muita comunicação entre CxProlog e Java.

O fator decisivo que levou à escolha das bibliotecas dinâmicas em vez de *sockets* foi a granularidade fina requerida ao nível da comunicação entre CxProlog e Java. A implementação dum protocolo de comunicação que possibilitasse o nível granularidade requerido pelo projeto seria árdua e demorada. Sem esquecer o facto de que o IDE a ser desenvolvido não ambiciona integrar outras implementações do Prolog, mas sim um IDE o mais próximo possível do CxProlog.

Neste ponto é importante realçar que o CxProlog é implementado em C, sendo necessária a utilização de métodos nativos para que o Eclipse/Java possa comunicar com a biblioteca dinâmica do CxProlog.

Eis um exemplo dum método nativo em Java, que foi utilizado ao longo do projeto:

```
/* CALL PROLOG from Java */
    public native static synchronized boolean CallProlog(String term) ;
```

Assim o projeto envolverá três linguagens no seu desenvolvimento:

- **Java** – Para desenvolver toda a parte gráfica do IDE baseado em Eclipse.
- **CxProlog** – Para desenvolver toda a lógica do IDE.
- **C** – Para modificações ao CxProlog.

Importante realçar que a implementação do CxIDE foi realizada principalmente em torno de uma biblioteca dinâmica do CxProlog, porém também foram utilizados sockets na implementação de uma das consolas (o CxIDE viria a usufruir de duas consolas (secção 7.6)).

6.3 Proposta de implementação é fazível?

Decidida a utilização da biblioteca dinâmica do CxProlog para o desenvolvimento do *plugin* CxIDE para Eclipse, surgia agora uma nova questão. Será possível utilizar bibliotecas dinâmicas em *plugins* para Eclipse?

Existem vários IDEs disponíveis como *plugins* para Eclipse, mas todos os encontrados estavam dependentes duma instalação local da linguagem que suportam. Por exemplo o PDT está dependente duma instalação local do SWI-Prolog utilizando depois *sockets* para comunicação entre *plugin* e SWI-Prolog.

Era muito importante esclarecer a dúvida em causa, pois caso não fosse possível a utilização de bibliotecas dinâmicas em *plugins* Eclipse a proposta de implementação apresentada seria impossível de realizar.

Foi decidido criar um *plugin* simples que utilizasse a biblioteca dinâmica do CxProlog. O sucesso deste simples *plugin* decidiria o futuro do projeto.

6.3.1 Plugin com interoperabilidade

Numa secção anterior (*secção 5.2*) foi explicado o desenvolvimento de um *plugin* denominado *com.ramos.plugin.first*. No entanto, o exemplo anterior apenas envolveu Java. No desenvolvimento dum IDE baseado em Eclipse para uma linguagem diferente de Java, poderá ser necessário existir algum mecanismo de interoperabilidade entre o Java e a linguagem que o IDE suportará, no caso do projeto proposto o CxProlog. Assim nesta secção continuar-se-á a utilizar o *com.ramos.plugin.first*

(*secção 5.2*) mas com uma diferença, o CxProlog será introduzido de modo a especificar o comportamento dos novos itens (botão e menu).

Primeiro, como é que o *plugin* pode aceder ao CxProlog? Como já descrito anteriormente (secção 6.1), foram ponderadas duas possibilidades: por *sockets* ou por acesso a uma biblioteca dinâmica.

Resumidamente, a escolha recaiu sobre as bibliotecas dinâmicas principalmente porque a granularidade de comunicação requerida (fina), é mais facilmente alcançável através da utilização de bibliotecas dinâmicas. Para além disso, enquanto o uso de *sockets* requeria uma comunicação entre o *plugin* e um processo de CxProlog, o uso duma biblioteca dinâmica possibilita a inclusão dessa mesma biblioteca internamente no *plugin* não sendo necessário contactar qualquer processo externo (maior integração).

O CxProlog possibilita a geração de uma biblioteca dinâmica através do comando **make lib**, sendo gerada a biblioteca **libcxprolog.so** em Linux. O ficheiro gerado pode ser adicionado à diretoria do *plugin* e uma forma de o carregar utilizando o Java é **System.loadLibrary(String libPath) .** Como a biblioteca gerada é implementada em C[21] é necessária a utilização de métodos nativos para que o Java possa comunicar com ela.

O CxProlog já está preparado para a interação com Java, disponibilizando o *Prolog.jar* para que o Java possa invocar métodos do CxProlog. Existem dois métodos interessantes que o *Prolog.jar* disponibiliza. O primeiro é o *boolean* **Prolog.CallProlog(String term)** que permite ao Java executar um termo na biblioteca dinâmica do CxProlog obtendo *true* caso o termo tenha sucesso e *false* em caso negativo, para além disso o resultado ou erro proveniente do termo é enviado para os canais de saída definidos. O segundo é o **Prolog.PostEvent(Object... elems)** que possibilita ao Java enviar um evento para uma fila de espera do CxProlog, sendo esse evento tratado assim que possível.

Por outro lado para o CxProlog chamar o Java existe o predicado :

```
java_call(+ClassOrInstance, +MethodSignature, +ArgList, -Result)
```

Este invoca um método Java ou construtor.

Os *plugins* em Eclipse tem de ser autossuficientes ou depender de outros *plugins*, por isso não é possível exportar *jars* externos (ex: Prolog.jar) para um projeto *plugin*. Para resolver o problema anterior o PDE oferece a possibilidade de criar um *plugin* a partir de um jar:

File → New → Other → Plug-in Development → Plug-in from existing jar archives

O *plugin* gerado a partir do *Prolog.jar* foi denominado *com.prolog.jar*. Após a criação da dependência entre o *com.ramos.plugin.first* e o *com.prolog.jar* toda a estrutura necessária para a utilização da biblioteca dinâmica do CxProlog estava montada (figura 16).

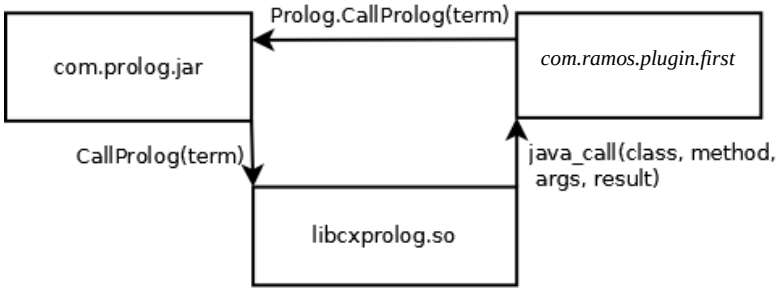


Figura 16 – Comunicação entre *plugins* e biblioteca dinâmica do CxProlog

O objetivo do exemplo é dar o poder de decisão ao CxProlog sobre o que fazer aquando da interação com os novos elementos da interface, implementados pelo *com.ramos.plugin.first*, para isso apenas é necessário modificar a classe **SampleHandler**. As modificações realizadas à classe **SampleHandler** (abaixo) possibilitam que o *plugin* informe o CxProlog quando um utilizador

[21] O CxProlog é implementado em C

interagiu com os novos elementos, em seguida o CxProlog informa o Java do que o mesmo deve fazer para responder à interação, neste caso executar o método **createDialog** que cria uma janela de diálogo. O método **Prolog.StartProlog()** proveniente do *com.prolog.jar* realiza o carregamento da biblioteca dinâmica **libcxprolog.so** em memória.

Eis a classe Java *SampleHandler* com comunicação entre Java e CxProlog:

```
public class SampleHandler extends AbstractHandler {

    static IWorkbenchWindow window;

    public Object execute(ExecutionEvent event) throws ExecutionException {
         window = HandlerUtil.getActiveWorkbenchWindowChecked(event);

        Prolog.StartProlog();
        Prolog.CallProlog("java_call('com/simple/plugin/handlers/SampleHandler','createDialog:()V',[],_)");

        return null;
    }

    public static void createDialog(){
        MessageDialog.openInformation(
                window.getShell(),
                "Plugin",
                "Hello, Eclipse/CxProlog world");
    }
}
```

O exemplo prático foi um sucesso (o *plugin* comportou-se como o esperado) sendo demonstrado que na implementação de um *plugin*, é possível o Java comunicar com o CxProlog (através do **Prolog.CallProlog**) e vice-versa (**java_call**). Também é demonstrado que o CxProlog tem acesso aos recursos disponíveis no Eclipse.

As conclusões deste exemplo foram muito importantes pois demonstram que é possível a escrita de *plugins* usando código misto, Java e CxProlog, tal como a utilização da biblioteca dinâmica por *plugins*.

6.4 Funcionalidades a implementar

No mínimo o CxIDE deveria possibilitar ao seu utilizador a edição e execução de ficheiros CxProlog. Assim era imprescindível ter um editor e uma consola. Obviamente o projeto ambicionava mais. De seguida a tabela de todas as funcionalidades que foram planeadas implementar no CxIDE:

Componente	Funcionalidade	Breve descrição
Editor	Realce de sintaxe	Coloração de texto diferente para: predicados *builtin*, comentários, variáveis, e outras categorias de termos.
	Validação sintática	Ser possível ao utilizador verificar a correção da sintaxe dum ficheiro CxProlog.
	Realce de erros	Os erros provenientes da validação sintática identificados com um sublinhar da zona afetada.
	Navegação no código	Abrir declaração ou ficheiros de origem de certos termos. Encontrar termos específicos
	Colapso e Expansão	Possibilidade de colapsar certas zonas do ficheiro em edição.
	Completação automática	O CxProlog têm vários predicados *builtin*, alguns com o mesmo funtor mas com diferentes aridades, por isso, será muito útil ter uma pronta lista de sugestões ao utilizador

Consola	Semelhante ao terminal	Uma consola, cuja sua interação seja o mais semelhante possível à execução do CxProlog num terminal.
	Sistema *toplevel*	Um sistema *toplevel* recebe *input* do utilizador, avalia o mesmo e retorna ao utilizador o resultado.
	Opera sobre a biblioteca dinâmica do *plugin*	É muito importante que a consola execute sobre a biblioteca dinâmica, pois só assim será possível dinamicamente estender o CxIDE e usar as anteriores configurações do utilizador. Estes aspetos têm uma profunda discussão na secção 8.6.2.
Navegador de projetos	Noção de projeto CxProlog	Uma noção de projeto CxProlog significaria que os projetos CxProlog teriam uma identidade própria. Assim esses projetos são facilmente identificáveis e podem ter opções específicas associadas.
	Noção de ficheiro CxProlog	O mesmo que a noção de projeto, mas para ficheiros. Ficheiro CxProlog poderiam ser identificados com a extensão ".cx".
Outras	Vista Estruturada	Vista estruturada sobre o atual ficheiro CxProlog em edição.
	Grafo de chamadas	Representação visual das chamadas no atual ficheiro CxProlog em edição.
	Perspetiva CxProlog	Uma perspetiva própria do CxIDE no Eclipse, que englobe o editor, navegador de projetos, vista estruturada e consola, descritos anteriormente.
	Preferências	Preferências do CxIDE, que permitam ao utilizador configurar definições sobre o mesmo.

Tabela 1 - Plano de funcionalidades a implementar no CxIDE

De realçar que as funcionalidades acima descritas constituíram o plano inicial do projeto. Ao longo da implementação surgiram dificuldades e oportunidades que levaram, respetivamente, à remoção e adição de funcionalidades à tabela acima.

Existiram algumas funcionalidades, cuja sua possível implementação ainda foi ponderada, uma dessas funcionalidades foi o depurador. Apesar de o depurador ser uma funcionalidade relativamente comum no mundo dos IDEs a sua implementação no CxIDE foi descartada. Os depuradores de Prolog são complicados devido ao *backtracking* e a sua implementação seria muito demorada, foi por isso decidido dar primazia a outras funcionalidades cuja sua implementação era vista como mais realista.

6.5 Funcionamento geral da solução proposta

Demonstrada a viabilidade da solução proposta para o projeto (secção *6.3.1*), é possível agora apresentar o funcionamento geral do IDE (figura 17):

1. O sistema arranca do lado do Java.

2. Ainda do lado do Java é carregada a biblioteca dinâmica do CxProlog.

3. Seguidamente o controlo é passado ao CxProlog, através do ***Prolog.CallProlog***

4. O CxProlog fica à espera de eventos do IDE. Por exemplo quando o utilizador interage com um comando (que foi implementado recorrendo a CxProlog), o CxProlog é notificado e definirá o comportamento adequado.

5. O IDE estará agora disponível para utilização, sendo que o Java se encarrega da parte gráfica relacionada com o Eclipse, e o CxProlog da parte lógica do funcionamento.

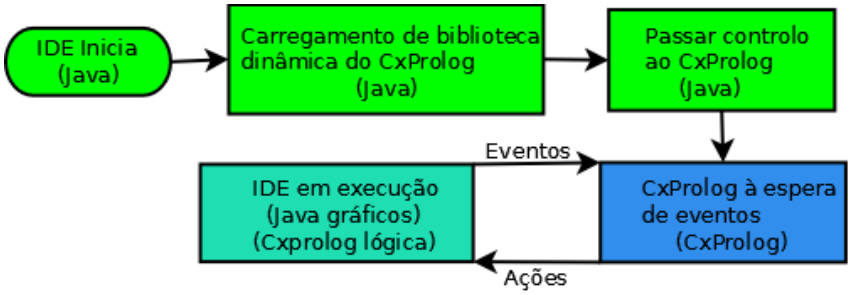


Figura 17 – Funcionamento da implementação do IDE para CxProlog

6.6 Troca de dados entre CxProlog e Java

Durante a implementação do projeto seria necessário trocar dados entre Java e CxProlog. Por exemplo, o Java necessita de obter os predicados *builtin* do CxProlog para implementar o realce de sintaxe (secção 4.1.1).

Durante a execução do Eclipse, o Java poderá necessitar de dados do CxProlog. Para possibilitar essa interação seriam seguidos os seguintes passos:

1. No Java, criar um método estático que atualize uma ou várias variáveis estáticas. Essas variáveis serão utilizadas para armazenar os dados pretendidos do CxProlog.
2. No CxProlog, criar um predicado que utilize *java_call/2* para invocar o método Java definido no passo **1**.
3. No Java, realizar um *CallProlog* para executar o predicado definido no passo **2**.

A figura 18 revela um exemplo da troca de dados entre CxProlog e Java. Concretamente, como o Java obtém a lista de predicados *builtin* do CxProlog.

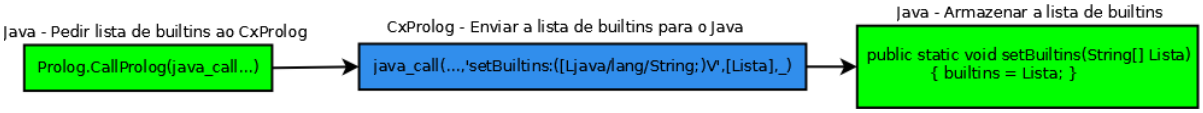


Figura 18 – Exemplo de troca de dados entre Java e CxProlog

7 Funcionalidades do CxIDE

Nesta secção são apresentadas de forma breve todas as funcionalidades que o CxIDE oferece aos seus utilizadores. O leitor encontrará no anexo 2 o manual de utilizador que descreve sucintamente os passos necessários para utilizar todas estas funcionalidades.

7.1 CxEditor

Um dos componentes mais interessantes do CxIDE, e onde o utilizador passará grande parte do seu tempo, é o editor. O editor do CxIDE é denominado **CxEditor** e suporta a edição de ficheiros CxProlog.

7.1.1 Análise sintática

O CxEditor realiza a análise sintática a um ficheiro durante a edição do mesmo. Os erros são realçados com o característico sublinhado vermelho e o utilizador também terá acesso a informações sobre a causa do erro. Para além disso, os erros detetados também serão visíveis nas vistas "*Problems View"* e "*Project Explorer*" (vistas padrão do Eclipse), permitindo ao utilizador do CxIDE uma fácil identificação de todos os erros no *workspace* que ainda não foram corrigidos.

7.1.2 Realce de sintaxe

O CxEditor suporta realce de sintaxe (baseado apenas na cor) que por omissão realça os seguintes elementos sintáticos:

- **Comentários**
- **Strings**
- **Variáveis**
- **Predicados *builtin***
- **Texto por omissão**

A figura 19 exemplifica todos os casos de realce de sintaxe.

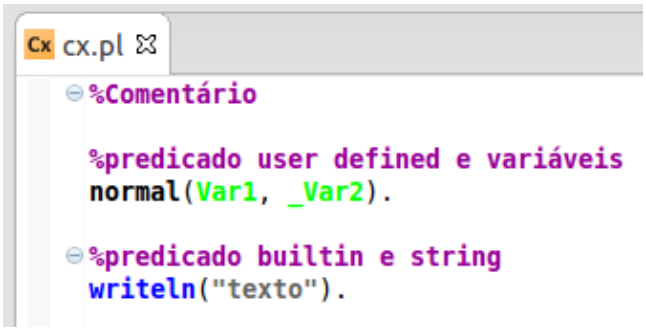


Figura 19 – Realce de sintaxe do CxIDE

Existe igualmente uma página de preferências (figura 20) dedicada à alteração da coloração do realce de sintaxe.

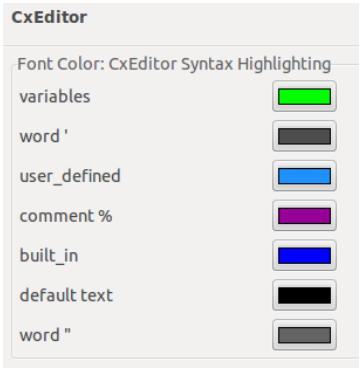


Figura 20 – Página de preferências do CxEditor

O CxIDE oferece ao CxProlog o controlo completo sobre o realce de sintaxe: tanto a especificação dos *tokens*, como a respetiva cor.

Para a definição do realce de sintaxe a partir do Prolog, estão disponíveis os seguintes predicados:

editor_singleLineRule(FieldName, StartSeq, EndSeq, EscChar, R, G, B)

Realça com a coloração **R**,**G**,**B** as linhas que iniciam com a sequência inicial **startSeq** e terminam com a sequência **EndSeq**. Poderá ser utilizado o caráter de escape **EscChar** nas sequências anteriores. **FieldName** é uma etiqueta (utilizada nas preferências) que fica associada ao tipo de *token*.

Numa das páginas de preferências do **CxEditor** aparece um painel que permite ao utilizador alterar a cor associada a qualquer tipo de *token*. Nesse painel são apresentadas as etiquetas de todos os tipos de *tokens* existentes seguidas de caixas de mudança de cor.

Exemplo:

Definição do realce de texto entre aspas:

```
:-editor_singleLineRule('word "', '"', '"', '\\', 100, 100, 100).
```

editor_endLineRule(FieldName, StartSeq, R, G, B)

Realça com a coloração **R**,**G**,**B** as linhas que iniciam com a sequência inicial **startSeq**.

Exemplo:

Definição do realce de comentários que são identificados por linhas que comecem com %

```
:-editor_endLineRule('comment %','%', 150, 0, 150).
```

editor_varRule(FieldName, R, G, B)

Realça com a coloração **R**,**G**,**B** as variáveis CxProlog.

Exemplo:

Definição do realce de sintaxe das variáveis CxProlog:

```
:-editor_varRule('variables',0,255,0).
```

editor_addNormalTextRule(FieldName, R, G, B)

Realça com a coloração **R**,**G**,**B** o texto padrão, ou seja todo o texto que não se enquandra em nenhuma das outras regras.

Exemplo:

Definição do realce de sintaxe do texto padrão do CxProlog

```
:-editor_addNormalTextRule('default text',0,0,0).
```

editor_addWordRule(FieldName,Word, R, G, B)

Realça com a coloração **R**,**G**,**B** a palavra **Word**.

Exemplo:

Definição do realce de sintaxe da palavra cx.

```
:-editor_addWordRule('Special','cx',200,200,0).
```

editor_addWordsRule(FieldName, List, R, G, B)

Realça com a coloração **R**,**G**,**B** as palavras na lista **List**.

Exemplo:

Definição do realce de sintaxe das palavras: cx, cxprolog e prolog.

```
:-editor_addWordRule('Special','[cx,cxprolog,prolog]',200,200,0).
```

editor_addPropertyHighlight(Pro, R, G, B)

Realça com a coloração **R**,**G**,**B** os predicados CxProlog com a propriedade **Pro**.

Exemplo:

Definição do realce de sintaxe de predicados com a propriedade *builtin* do CxProlog.

```
:-editor_addPropertyHighlight('built_in',0,0,255).
```

O realce de sintaxe pode ser definido pelo utilizador através da edição do ficheiro *config.pl* usando os predicados acima descritos. Para editar o ficheiro config.pl é preciso usar o predicado *config/0*, o qual acede a um recurso guardado num ficheiro Jar e abre uma janela de edição. É suposto os *plugins* para Eclipse funcionarem mesmo assim.

Nada também impede o utilizador de usar diretamente os predicados anteriores na **Consola Interna** (secção 7.6).

7.1.3 Completação automática

O CxIDE suporta completação automática (figura 21). As propostas de completação são as assinaturas dos predicados *builtin* do CxProlog, sendo também mostrada uma descrição dos mesmos. A completação automática é baseada no manual do CxProlog. Assim o utilizador tem acesso a uma descrição o mais completa possível de todas as propostas.

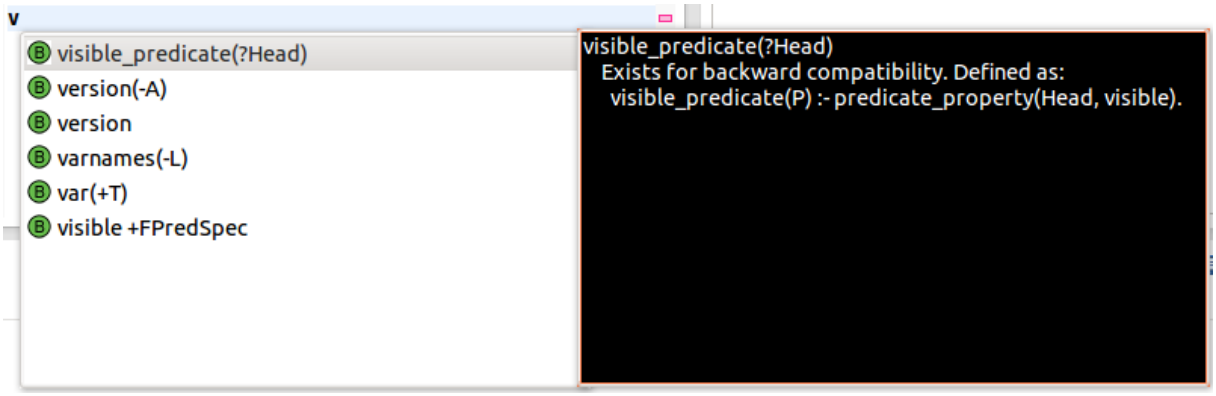


Figura 21 – Completação automática do CxIDE

7.1.4 Colapso e Expansão

O **CxEditor** suporta colapso e expansão de cláusulas (figura 22). Assim o utilizador tem a possibilidade de manter a visualização do seu ficheiro em edição o mais simples possível.

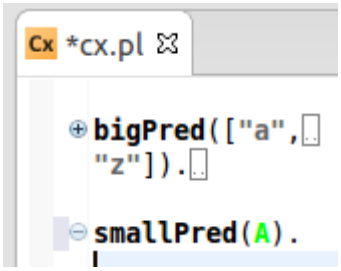


Figura 22 – Colapso e expansão do CxEditor

7.1.5 Injeção de texto na consola

A seleção dum excerto de código no **CxEditor** e a sua injeção na consola interna para execução imediata (ao premir **F6**) é uma possibilidade. Esta funcionalidade é muito útil, especialmente quando se pensa no CxIDE também como um ambiente de teste e execução de programas.

7.1.6 Menu de contexto do CxEditor

O **CxEditor** suporta um menu de contexto próprio. Portanto, quando o utilizador premir o botão da direita do rato sobre a janela de edição, surgirá um menu de *popup* que é o referido menu de contexto.

Este menu de contexto tem um conteúdo fixo, que pode ser observado na figura 23. Entre os vários itens do menu existe um especial chamado "CxProlog" que pode ser controlado programaticamente.

O item "CxProlog" tem um submenu associado. O utilizador pode adicionar itens a esse submenu utilizando o seguinte predicado:

```
add_contextItem(MenuName,CMD)
```

O predicado acima adiciona um novo item ao menu de contexto do CxEditor. O novo item terá o título **MenuName** e o seu comportamento é definido pelo comando CxProlog CMD.

Eis um exemplo:

```
:-add_contextItem('Inject selected text','editor_injectSelCode')
```

Este comando adiciona um novo item ao menu de contexto do CxEditor (figura 23), onde **editor_injectSelCode** é um golo que só o CxIDE conhece. Existe uma pequena coleção de predicados auxiliares de apoio ao IDE descritos no anexo 3.

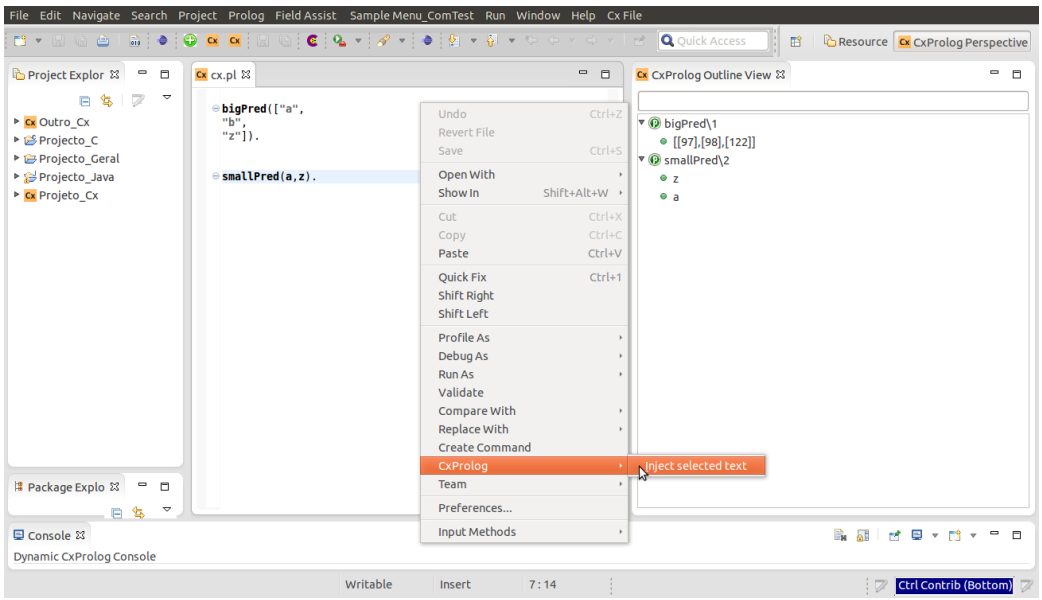


Figura 23 – Resultado da adição dum item ao menu de contexto

Vista Estruturada

A vista estruturada do CxProlog (**CxProlog Outline**) oferece uma representação abstrata do ficheiro em edição no **CxEditor**. Essa representação inclui cláusulas e átomos. Também é fornecida uma forma rápida de navegação. Através de duplo clique num dos elementos da vista estruturada é automaticamente realçada e revelada a sua localização no **CxEditor** (figura 24).

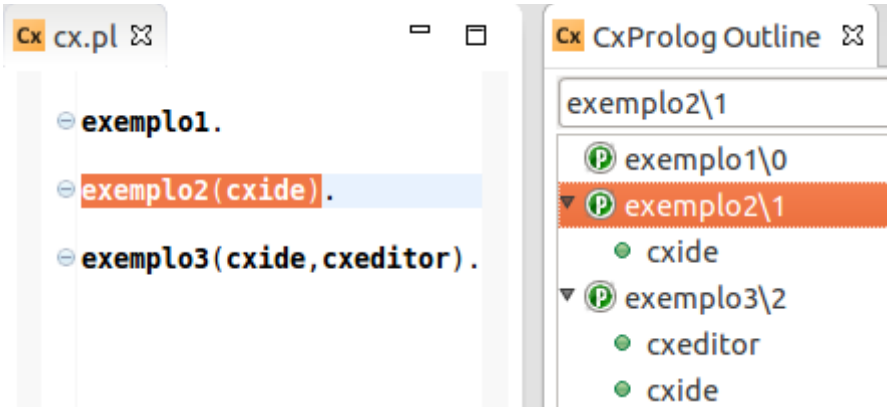


Figura 24 – Vista estruturada do CxIDE (à direita)

7.2 Consolas

O CxIDE oferece dois tipos consolas (figura 25), que o utilizador poderá escolher:

Consola Interna (CxProlog Internal Console) → Consola que opera sobre uma instância do CxProlog que está embebida no próprio *plugin* CxIDE. Usar esta consola é preferível na maior parte das situações porque é a única que permite a execução de golos que atuam sobre o CxIDE (anexo 3). É também a única que permite observar o *output* de golos que sejam lançados através de menus e diálogos.

Consola Externa (CxProlog External Console) → Consola que opera sobre uma instalação externa ao CxIDE através dum *socket*. Esta consola não tem qualquer conexão com o ambiente gráfico do CxIDE, o que significa que não pode atuar sobre ele. A vantagem desta consola é que permite ao utilizador trabalhar com uma versão de CxProlog diferente da que esta embebida no *plugin*. Sendo inclusive possível ligar a consola a um servidor local de CxProlog (figura 26). Também é possível ligar a um servidor remoto, por ventura mais poderoso, mas com algumas lacunas explícitas na secção 8.5.1. Bem entendido, o Prolog embebido continua sempre ativo a controlar a parte gráfica do CxIDE.

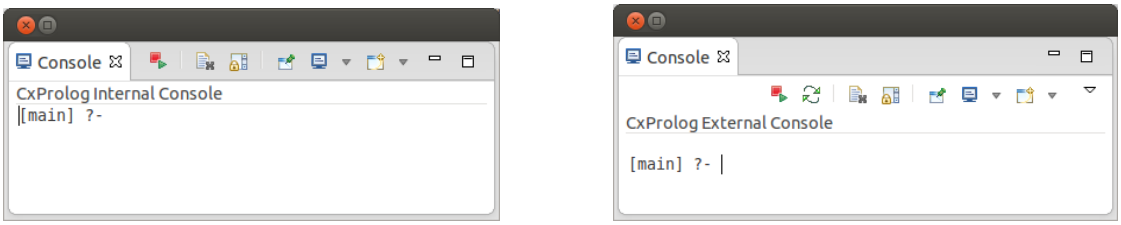


Figura 25 – Diferentes consolas do CxIDE

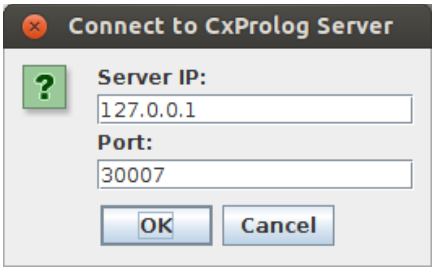


Figura 26 – Ligação da consola a um servidor de Prolog

7.3 Preferências

O CxIDE tem o seu próprio menu nas preferências do Eclipse, denominado CxProlog (figura 27). No Eclipse uma página de preferências é um diálogo para modificar determinados valores dum componente. As preferências do CxIDE são compostas por duas páginas de preferências:

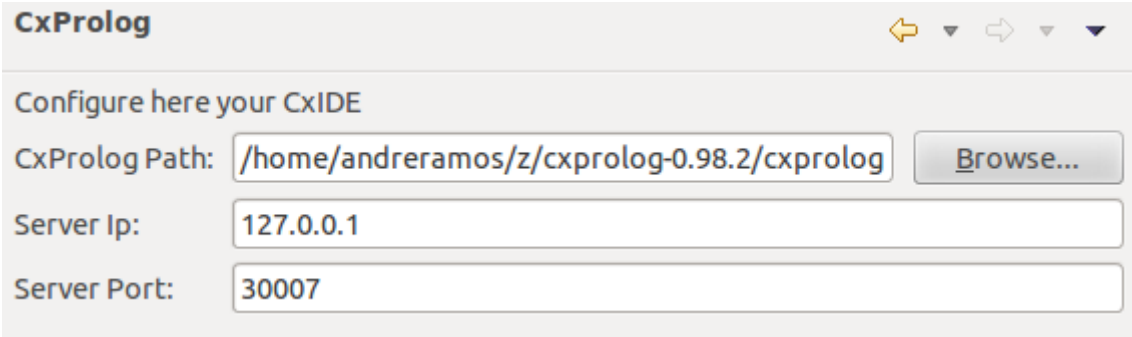


Figura 27 – Página de preferências CxProlog do CxIDE

Página CxProlog: Permite alterar a instalação local do CxProlog utilizada na consola externa (secção 7.2), tal como alterar definições relacionadas com o servidor de CxProlog a ser utilizado.

Página CxEditor: Esta página de preferências que permite alterar a coloração do realce de sintaxe, já foi descrita na secção 7.1.2.

7.4 Navegador de projetos

Seguindo o exemplo dos outros IDEs baseados em Eclipse, o CxIDE utiliza o navegador de projetos padrão do Eclipse (figura 28).

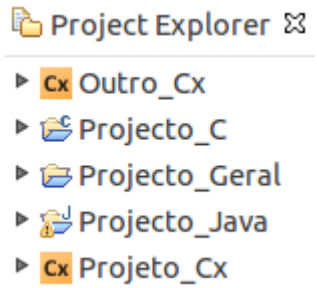


Figura 28 – Navegador de projetos

7.4.1 Natureza de projeto CxProlog

Existe uma noção de projeto CxProlog. Tal noção permite a distinção inequívoca dos projetos CxProlog de todos os outros projetos existentes no ambiente de trabalho do Eclipse.

Essa distinção é:

- **Visual** - utilização de distintos ícones, na representação visual de projetos CxProlog.
- **Funcional** - menus de contexto e comandos apenas específicos aos projetos CxProlog.
- **Identidade própria** - Cada noção de projeto detêm um identificador único. Esse identificador é usado internamente para distinguir inequivocamente os diferentes tipos de projetos existentes.

7.4.2 Criação de projetos e ficheiros CxProlog

É possível criar um projetos CxProlog em Eclipse da seguinte forma:

File→New→Other→Other→CxProlog Project

O utilizador poderá então utilizar o *Wizard* de criação de projetos CxProlog. IDEs baseados em Eclipse tendem a oferecer *wizards* para a criação dos seus projetos específicos. O CxIDE não fica atrás e oferece aos seus utilizadores um *wizard* para uma fácil e intuitiva criação de projetos CxProlog (figura 24).

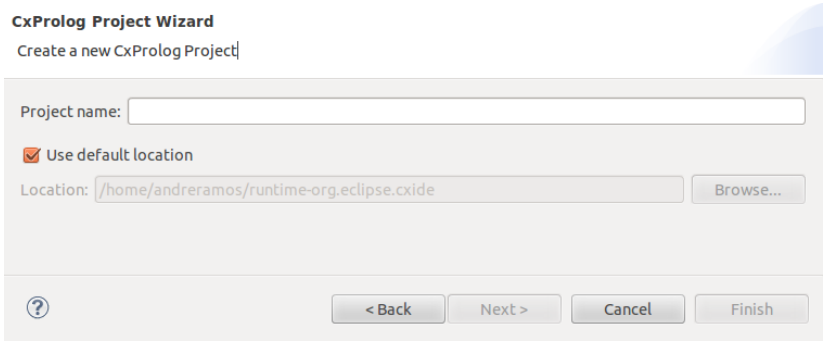


Figura 29 – Wizard para a criação de projetos CxProlog

De forma semelhante à criação de projetos, para criar um ficheiro CxProlog o utilizador poderá ir ao seguinte menu:

File→New→Other→Other→CxProlog File

7.4.3 Conversão de projetos

Através dos menus do Eclipse, o utilizador tem sempre a possibilidade de criar um projeto geral, não associado a qualquer linguagem ou *plugin* específico.

É recorrente a utilização de projetos gerais em Eclipse para agregar de forma *ad-hoc* um conjunto de ficheiros em que se está a trabalhar. O CxIDE oferece a possibilidade de converter um projeto geral com para um projeto CxProlog. Os projetos que não sejam CxProlog, incluem o item "Covert to CxProject" no seu menu de contexto (figura 30).

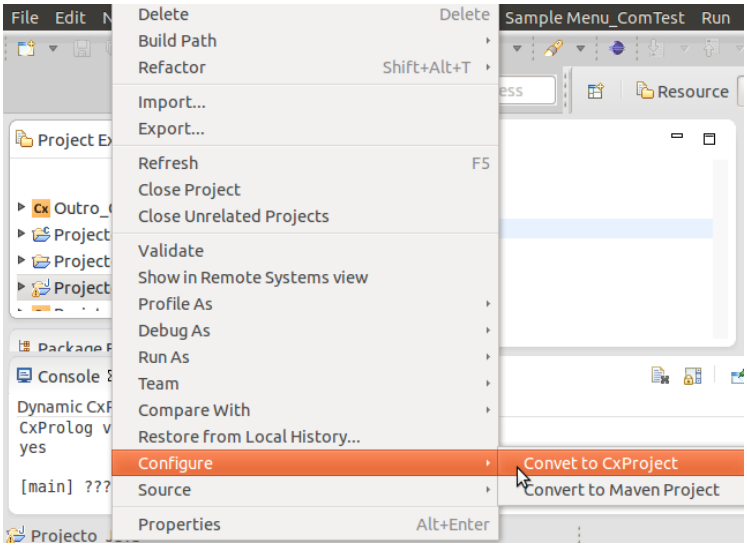


Figura 30 – Conversão para projeto CxProlog

7.5 Extensão e configuração

Um dos principais objetivos do CxIDE é o mesmo seja extensível e configurável via CxProlog

Para configurar o CxIDE o utilizador deverá editar o ficheiro *config.pl*. O ficheiro *config.pl* define as configurações iniciais do CxIDE, o seu estado inicial é revelado no anexo 5. O utilizador poderá alterar ou remover estas configurações mas fará isso à sua responsabilidade. Para editar o ficheiro *config.pl* o utilizador precisa de executar o golo "*config*" na consola interna do CxIDE. Assim será automaticamente aberto o ficheiro *config.pl* e as alterações ao mesmo estarão disponíveis após a reiniciação do Eclipse.

O utilizador também pode realizar extensões ou configurações durante uma sessão de trabalho com o CxIDE, sendo estas realizadas ao executar golos adequados (anexo 3) diretamente na consola interna (figura 31) do CxIDE ou após consulta dum ficheiro contendo comandos nessa mesma consola. As extensões e configurações realizadas desta forma não são persistentes. Isto é, apenas têm efeito durante a execução em causa do CxIDE. As figuras 30 e 31 revelam o exemplo da execução de um golo que cria um novo menu durante a execução do CxIDE.

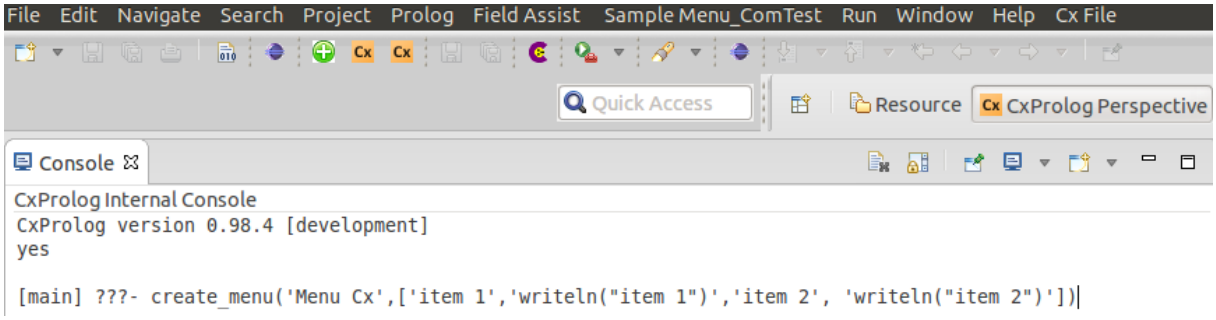


Figura 31 – Execução dum golo na consola interna que adiciona um novo menu

O resultado terá efeitos imediatos e o ambiente incluirá a nova alteração (figura 32).

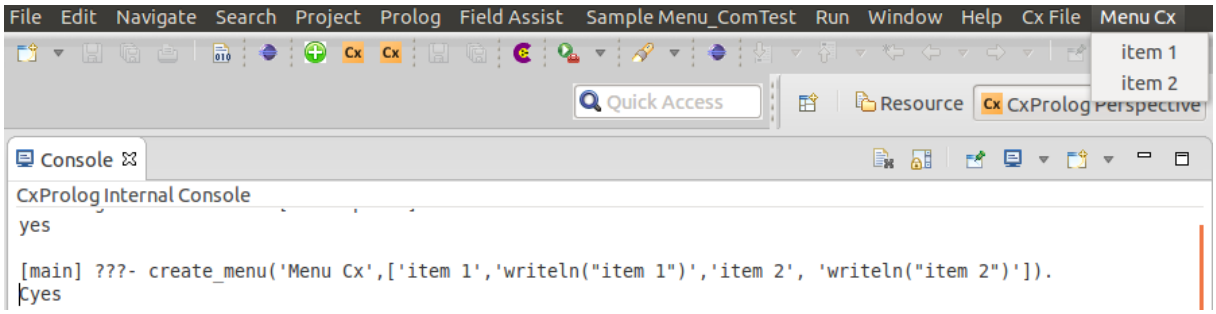


Figura 32 – Resultado da execução dum golo para criar um novo menu

7.5.1 Criação de menus

Esta secção descreve sucintamente os predicados disponibilizados pelo CxIDE para a criação, alteração e remoção de menus. Na secção 8.3 é descrita a implementação e revelados exemplos de utilização desses predicados.

Para trabalhar com menus o utilizador do CxIDE poderá recorrer ao ficheiro *config.pl* ou a consola interna (ver secção 7.2) usando os predicados adequados. Eis a lista dos predicados que podem ser usados:

- **`create_menu(NomeMenu)`**
  Adiciona um novo menu, caso não exista, com o nome **NomeMenu** à barra principal de menus do Eclipse. Caso já exista um menu **NomeMenu** a adição não será realizada e o

utilizador será notificado.

- **`create_menu(NomeMenu, Item, CMD)`**
  Adiciona um novo item com o nome **Item** ao menu com o **NomeMenu**. O menu **NomeMenu** será criado caso não exista. O comando CxProlog **CMD** define o comportamento do item, isto é, quando o utilizador selecionar esse item será executado o comando dado.

- **`create_menu(NomeMenu, Items)`**
  Adiciona um novo menu, caso não exista, com o nome **NomeMenu**. A esse menu serão adicionados os itens e respetivos comportamentos descritos na lista **Items**.

- **`delete_menu(NomeMenu)`**
  Elimina um menu com o nome **NomeMenu**.

- **`listMenus`**
  Revela um diálogo com a lista de todos os menus criados através do CxProlog.

7.5.2 Criação de diálogos

Esta secção descreve sucintamente os predicados disponibilizados pelo CxIDE para a criação e alteração de diálogos. Na secção 8.3.4 é descrita a implementação e revelados exemplos de utilização desses predicados.

Eis a lista dos predicados disponíveis para trabalhar com diálogos:

- **`create_dialog(Titulo)`**
  Cria um simples diálogo com o título **Titulo** e que apresenta dois botões "OK" e "CANCEL". Ambos os botões têm o mesmo comportamento que é simplesmente fechar o diálogo. O diálogo criado é não bloqueante, ou seja, durante a sua invocação o utilizador conseguirá interagir com o Eclipse.

- **`dialog_setAction(Titulo, CMD)`**
  Define o comportamento do botão "OK" do diálogo com o título **Titulo**. O comportamento será executar o comando CxProlog **CMD**.

- **`add_textForm_dialog(Titulo, Label)`**
  Adiciona um campo de edição de texto ao diálogo com o título **Titulo**. O campo adicionado será identificado pela etiqueta **Label**.

- **`add_numForm_dialog(Titulo, Label)`**
  Adiciona um campo de edição de numéricos ao diálogo com o título **Titulo**. O campo adicionado será identificado pela etiqueta **Label**.

- **`getTextField(Titulo, Label,V)`**
  Obtém o valor na variável **V** do campo de texto com a etiqueta **Label** do diálogo com o título **Titulo**.

- **`getNumField(Titulo, Label, N)`**
  Obtém o valor na variável **N** do campo de numéricos com a etiqueta **Label** do diálogo com o título **Titulo**.

- **dialog_setOutput(Titulo, [Out1 |OutN])**
  Exibe um diálogo, com o título **Titulo**, que apresenta todos os elementos da lista declarada (2º parâmetro). O diálogo revelado é bloqueante, ou seja, o utilizador só poderá interagir com o Eclipse quando fechar o diálogo.

- **show_dialog(Titulo)**
  Exibe o diálogo com o título **Titulo**.

7.5.3 Definição do realce de sintaxe

Esta secção descreve sucintamente os predicados disponibilizados pelo CxIDE para a criação de regras de realce de sintaxe. Na secção 8.4.1 é descrita a implementação e revelados exemplos de utilização desses predicados. De destacar que cada vez que é criada uma nova regra de realce é adicionado um novo campo às preferências do CxEditor. Esse campo permite que o utilizador modifique durante a execução do CxIDE a coloração da sua sintaxe.

Eis a lista de predicados para definir regras de realce de sintaxe:

- **editor_singleLineRule(RuleName, StartSeq, EndSeq, EscChar, R, G, B)**
  Se uma linha começa pela sequência de carateres **StartSeq** e termina com a sequência de carateres **EndSeq**, então a mesma terá a coloração definida por **R**, **G**, **B**. É possível utilizar o caráter **EscChar** como escape.

- **editor_endLineRule(RuleName,StartSeq,R,G,B)**
  Se uma linha começa pela sequência de carateres **StartSeq**, então a mesma será colorida pela cor **R**, **G**, **B**.

- **editor_varRule(RuleName,R,G,B)**
  As variáveis Prolog serão coloridas pela cor **R**, **G**, **B**.

- **editor_addNormalTextRule(RuleName,R,G,B)**
  Todo o texto que não seja incluído nas outras regras de sintaxe será colorido pela cor **R**, **G**, **B**.

- **editor_addWordRule(RuleName,Word,R,G,B)**
  A sequência concreta de carateres **Word** será colorida de **R**, **G**, **B**.

- **editor_addWordsRule(RuleName,[Word1|WordN],R, G, B)**
  Todas as sequências concretas de carateres definidas na lista (2º parâmetro) serão coloridas de **R**, **G**, **B**.

- **editor_addPropertyHighlight(Pro,R,G,B)**
  Todos predicados CxProlog com a propriedade Pro serão coloridos de **R**, **G**, **B**.

7.5.4 Predicados utilitários

Esta secção descreve sucintamente os predicados disponibilizados pelo CxIDE. Alguns desses predicados permitem utilizar funcionalidades do Eclipse/Java (diálogo de seleção de diretorias), enquanto outros estão relacionados com o CxIDE (ex: config). A ambição destes predicados é facilitar ao utilizador do CxIDE o desenvolvimento de extensões.

- **`file_chooser(FilePath)`**
  Abre uma janela de diálogo para a navegação nas diretorias com o propósito de selecionar um ficheiro. O caminho absoluto do ficheiro selecionado é retornado na variável **FilePath**.

- **`dir_chooser(DirPath)`**
  Abre uma janela de diálogo para a navegação nas diretorias com o propósito de selecionar uma diretória. O caminho absoluto do diretório selecionado é retornado na variável **DirPath**.

- **`fd_chooser(Path)`**
  Abre uma janela de diálogo para a navegação nas diretorias com o propósito de selecionar uma diretória ou um ficheiro. O caminho absoluto do selecionado é retornado na variável **Path**.

- **`open_file(Path)`**
  Abre no CxEditor o ficheiro com o caminho absoluto **Path**.

- **`config`**
  Abre no CxEditor o ficheiro *config.pl* do CxIDE que permite configurar o mesmo.

- **`selectedText(Text)`**
  Obtém na variável **Text** o texto atualmente selecionado no CxEditor.

- **`editor_injectSelCode`**
  Injeta na consola ativa do CxIDE o texto atualmente selecionado no CxEditor.

- **`editor_setContentAssistFile(FilePath)`**
  O ficheiro XML no caminho **FilePath** definirá a completação automática do CxEditor.

8 Implementação

8.1 Estrutura de plugins

Esta secção descreve, sucintamente, os *plugins* que compõem o CxIDE e a estrutura dos mesmos.

O CxIDE é composto por dois *plugins* (figura 33): **org.eclipse.cxide** e o **org.prolog.jar**.

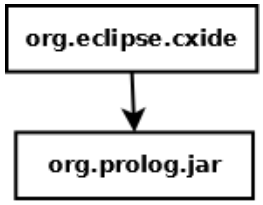


Figura 33 – Arquitetura *plugin* do CxIDE

org.prolog.jar - Responsável por possibilitar a interação com a biblioteca dinâmica do CxProlog.

org.eclipse.cxide – Engloba todos os componentes que constituem o CxIDE.

O **org.eclipse.cxide** é constituído por dez *packages* (figura 34) que representam diferentes componentes:

- **org.eclipse.cxide** (package) - Define o arranque do CxIDE. Aqui são realizadas as configurações inicias do CxIDE.
- **org.eclipse.cxide.console** – Código relacionado com as consolas do CxIDE.
- **org.eclipse.cxide.editor** – Todo o código relacionado com o CxEditor.
- **org.eclipse.cxide.handler** – Existem algumas extensões que são adicionadas ao Eclipse (como botão para consult) aqui é definido o comportamento de tais extensões.
- **org.eclipse.cxide.Menu_ops** – Código que permite a criação de menus e diálogos.
- **org.eclipse.cxide.perspective** – Definição da perspetiva do CxIDE.
- **org.eclipse.cxide.utilities** – Engloba várias funcionalidades que não eram específicas a um único componente.
- **org.eclipse.views** – Define as vistas que o CxIDE oferece.

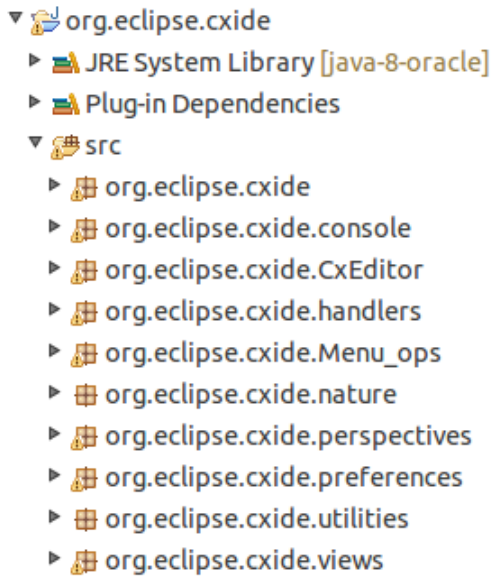


Figura 34 – Packages que constituem o CxIDE

O leitor interessado poderá consulta a arquitetura de outro IDE para CxProlog no anexo 1.

8.2 Configurações e extensões iniciais ao CxIDE

O utilizador do CxIDE disponibiliza duas distintas opções para configurar ou estender o CxIDE (ver secção 7.5).

Para configurar o CxIDE o utilizador deverá editar o ficheiro *config.pl*. No *config.pl* é esperada a utilização de comandos dedicados à extensão ou configuração do CxIDE. Esses comandos serão referidos como *CxPreds.* O leitor interessado poderá consultar o anexo 3, que engloba todos os CxPreds existentes. Em termos de implementação, era necessário assegurar que antes da consulta do *config.p*l a biblioteca dinâmica do CxProlog já tivesse conhecimento dos *CxPreds*. Existia a possibilidade tornar os *CxPreds* predicados *builtin* do CxProlog, mas seriam predicados *builtins* que só poderiam ser utilizados no CxIDE o que não faria sentido. A solução encontrada foi muito mais simples. Foi criado um novo ficheiro *root.pl* que definiria todos os *CxPreds*. No arranque do CxIDE, primeiro é consultado o *root.pl* de forma a "instalar" os *CxPreds* na biblioteca dinâmica (anexo 5), em segundo é consultado o *config.pl* para a configuração inicial do CxIDE.

Todos os *plugins* em Eclipse podem ter uma classe principal conhecida como **Activator**. Essa classe é responsável pelo ciclo de vida do *plugin*, sendo possível definir as ações iniciais ou finais do *plugin*.

Eis um excerto do Activator do CxIDE:

```
//Método da classe Activator que define as ações a realizar no arranque
//do plugin

public void start(BundleContext context) throws Exception {
   super.start(context);

   //Carregar biblioteca dinâmica
   prolog.Prolog.StartProlog();

   //Consultar ficheiro root que define os CxPreds
   prolog.Prolog.CallProlog("consult('root.pl')");

   //Consultar ficheiro config que configura o CxIDE
   prolog.Prolog.CallProlog("consult('config.pl')");

   //Lançar as duas consolas do IDE
   consoleInternal = new CxInternalConsole();
   consoleExternal = new CxExternalConsole();}
```

8.3 Estender o CxIDE usando o CxProlog

Nesta secção será descrita a forma de implementar o suporte para extensibilidade do CxIDE, através de código CxProlog.

Para estender o CxIDE com uma nova funcionalidade disponível do lado do CxProlog é necessário realizar realizar os seguintes passos:

1. **Desenvolver o lado Java da extensão programaticamente.**
   O PDE (Plugin Development Environment) possibilita a realização de simples extensões ao Eclipse usando um ambiente visual, por vezes sem ser necessário escrever uma única linha de código. O lado Java da extensão tem de ser realizado programaticamente, para posteriormente ser invocado pelo CxProlog.

2. **Testar se a extensão pode ser realizada durante a execução do CxIDE.**
   Algumas extensões ao Eclipse só funcionam na fase de arranque do *plugin*. Por exemplo é possível adicionar menus durante a execução do Eclipse, mas só é possível adicionar novos botões à barra de ferramentas na fase de arranque. Portanto, neste passo poderá ser necessário desistir.

3. **Desenvolver um método estático em Java de interface com o Prolog.**
   O CxProlog invocará métodos estáticos Java para realizar as extensões ao Eclipse.

4. **Criar um predicado CxProlog que utilize java_call/2 para invocar o método desenvolvido no passo 3.**
   A invocação do método Java pelo CxProlog é realizada através de *java_call/2*.

5. **Testar.**

   Iniciar o CxIDE, executar o predicado criado no passo 4 e determinar se funciona sem problemas.

8.3.1 Criação de menus

Seguiremos os passos acima apresentados para revelar como i implementada a criação de novos menus ao Eclipse utilizando o CxProlog.

1. **Desenvolver o lado Java da extensão programaticamente.**

Para adicionar menus programaticamente ao Eclipse através do Java é usada a classe **MenuManager**[22] da plataforma Eclipse.

Um **MenuManager** pode representar uma barra de menus, um submenu ou um menu de contexto. Assim um **MenuManager** pode ser constituído por várias outras instâncias de **MenuManager**.

Um menu é identificado por um nome que é exibido ao utilizador no ambiente de trabalho do Eclipse (ex: **File**) e um ID que é usado internamente para identificar esse menu. Podem existir dois menus com o mesmo nome mas não com o mesmo ID. Muitas vezes os menus têm o mesmo nome e ID para possibilitar ao utilizador uma fácil identificação do menu, mesmo internamente. Por exemplo o menu com o nome **File** tem igualmente **File** como ID.

Um menu pode ser utilizado simplesmente para organizar outros menus ou para possibilitar a execução dum comando. Por exemplo, o menu **Edit** do Eclipse não executa qualquer comando simplesmente é a raiz de outros menus como o **Copy** ou o **Paste**. Por outro lado o **Copy** e **Paste** são menus que executam comandos muito conhecidos por todos os utilizadores.

Seguidamente, será revelado um exemplo que esclarece a utilização do **MenuManager**.

A barra principal de menus do Eclipse (figura 35) é um **MenuManager** e portanto pode ser utilizada como alvo para novas adições.

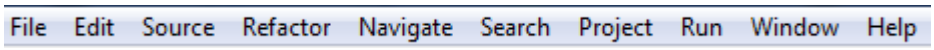


Figura 35 – Barra principal de menus do Eclipse

O seguinte excerto de código adiciona um novo menu à barra principal de menus do Eclipse:

```
//obter a janela atual do Eclipse
IWorkbenchWindow  window =
Workbench.getInstance().getActiveWorkbenchWindow();
//Obter a barra de menus principal do Eclipse
MenuManager menuManager = ((WorkbenchWindow)window).getMenuManager();
//Criar um novo menu com o nome e id "Novo Menu"
MenuManager menu = new MenuManager("Novo Menu", "Novo Menu");
//adicionar o novo menu
menuManager.add(menu);
//atualizar a barra principal de menus do eclipse para revelar o novo
//menu
```

[22] Mais informações sobre a classe MenuManager em: goo.gl/M4fipT

```
menuManager.update();
```

O código para adicionar um novo menu programaticamente ao Eclipse estava pronto.

Faltava agora testar se o mesmo pode ser executado e se tem os efeitos esperados durante a execução do Eclipse.

Para isso vamos voltar ao exemplo da secção 5.2 que implementa um *plugin* com um simples botão que permite executar algo aquando da interação. O código foi modificado para que seja adicionado um menu quando o utilizador clicar no botão (abaixo).

Eis o **Handler** que adiciona um novo menu à barra principal de menus do Eclipse:

```
public class SampleHandler extends AbstractHandler {

    public SampleHandler() {
    }

    public Object execute(ExecutionEvent event) throws ExecutionException {

        IWorkbenchWindow  window = Workbench.getInstance().getActiveWorkbenchWindow();
        MenuManager mainMenu = ((WorkbenchWindow)window).getMenuManager();

        MenuManager newMenu = new MenuManager("Novo Menu","Novo Menu");
        mainMenu.add(newMenu);
        mainMenu.update();
        return null;
    }
}
```

Era agora necessário testar se a adição poderia ser realizada durante a execução.

2. **Testar se a extensão pode ser realizada durante a execução do CxIDE.**

E o resultado foi um sucesso (figura 36). É possível dinamicamente (durante a execução) adicionar um novo menu ao Eclipse. No entanto a adição foi realizada usando Java. É necessário continuar o processo de desenvolvimento para que a adição do menu possa ser realizada através do CxProlog.

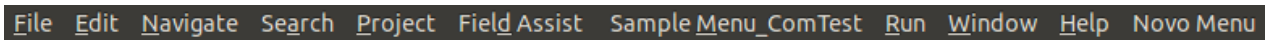


Figura 36 – Barra principal de menus do Eclipse com o "Novo Menu"

3. **Desenvolver um método estático em Java de interface com o Prolog.**

O *java_call/2* só pode ser utilizado sobre métodos estáticos, portanto é necessário criar um método estático em Java que permita a adição dum menu ao Eclipse.

Eis o método estático Java para a adição dum menu ao Eclipse:

```
Public static void createMenu(String menu_name){
//obter a janela atual do Eclipse
IWorkbenchWindow  window =
Workbench.getInstance().getActiveWorkbenchWindow();
//Obter a barra de menus principal do Eclipse
MenuManager menuManager = ((WorkbenchWindow)window).getMenuManager();
//Criar um novo menu com o nome e id iguais a "menu_name"
MenuManager menu = new MenuManager(menu_name, menu_name);
//adicionar o novo menu
menuManager.add(menu);
//atualizar a barra principal de menus do eclipse para revelar o novo
menu
menuManager.update();
}
```

Concluído o método estático, existe agora a possibilidade de usar o *java_call/2* para invocar o método acima.

4. **Criar um predicado CxProlog que utilize java_call/2 para invocar o método desenvolvido no passo 3.**

Foi criado um novo predicado que permite facilmente ao utilizador estender o Eclipse ao adicionar um novo menu à barra principal de menus do Eclipse.

```
create_menu(NomeMenu):-
java_call('org/eclipse/cxide/Menu_ops/Menu_Operations','createMenu:(Ljava/lang/String;)V',[NomeMenu],_).
```

Predicado concluído, só faltava testar.

5. **Testar.**

O teste realizado foi novamente um sucesso, sendo obtido o primeiro predicado Prolog que consegue estender o Eclipse, durante a sua execução. E será o primeiro predicado a entrar para o ficheiro *root.pl* (anexo 5).

Para adicionar um menu através do CxProlog ao Eclipse o utilizador (figura 37) pode usar o novo predicado **create_menu** no ficheiro *config.pl*, diretamente na consola interna do CxIDE que opera sobre a biblioteca dinâmica, ou até ao consultar um *ficheiro.pl* que utilize o **create_menu**.

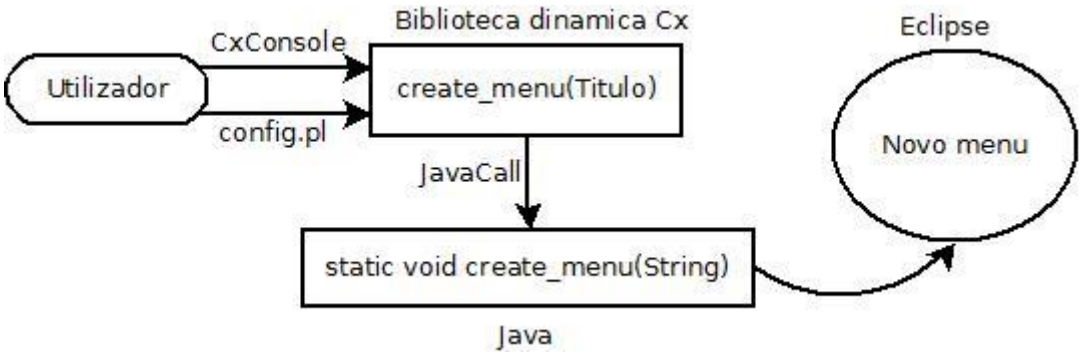


Figura 37 – Diagrama do processo de criação dum novo menu

8.3.2 Definição do comportamento dum menu

Na secção anterior foi explicado como se pode adicionar um menu ao Eclipse utilizando o CxProlog. Porém um menu sem qualquer submenu e sem nenhum comportamento é simplesmente inútil. É suposto, não só o utilizador conseguir criar um menu em CxProlog, como definir o seu comportamento, se assim o desejar.

Vamos modificar o código anterior com o objetivo de obter um predicado CxProlog que permita criar um menu e definir o seu comportamento usando CxProlog.

Um **MenuManager** pode ter uma ou mais ações (classe **Action** da plataforma Eclipse) associadas, estas ações permitem definir o comportamento do menu envolvido.

Eis como se pode definir uma ação em Java.

```
class ExampleAction extends Action {
        public About1() {
            super("Nome Acao", AS_PUSH_BUTTON);
        }
        public void run() {
      //Aqui é definido o comportamento da ação
            System.out.println("Ação ativada!!");
        } }
```

Mas no exemplo acima o comportamento da ação é definido através do Java. O exemplo é agora modificado para que o comportamento da ação seja definido através do CxProlog, usando o **CallProlog**.

```
private static Action createAction(String action_name, String action_cmd){
            final String action_Name = action_name;
            final String action_CMD = action_cmd;
            class PrologAction extends Action {
              public About1() {
                  super(action_Name, AS_PUSH_BUTTON);
              }
              public void run() {
                  Prolog.CallProlog(action_CMD);
              }
          } return new PrologAction();
      }
```

Com o código acima revelado é possível criar uma ação Java que executará um determinado comando Prolog.

```
createAction("Versão","version.")
```

É uma ação Java identificada pelo nome "Versão" que quando executada pediria a versão ao CxProlog.

Para criar um novo predicado que possibilite ao utilizador do CxIDE adicionar um novo menu ao Eclipse e definir o seu comportamento usando CxProlog, foi duplicado e ligeiramente alterado o código do **create_menu**.

```
public static void createMenu(String menu_name,String prolog_cmd){
          //O novo menu a ser adicionado
          MenuManager menu = new MenuManager(menu_name, menu_name);
          //A ação que define o comportamento do menu
          Action prologAction = createAction(prolog_cmd, prolog_cmd);
          //Adicionar a ação ao novo menu
          menu.add(prologAction);
          //O novo menu será sub-menu da barra principal de menus
        menuManager.add(menu);
        menuManager.update(); }
```

Seguidamente foi criado o predicado com o *java_call/2* que invocaria o método estático Java acima descrito.

```
create_menu(NomeMenu,PrologCmd):-
java_call('org/eclipse/cxide/Menu_ops/Menu_Operations','createMenu:(Ljava/l
ang/String;Ljava/lang/String;)V',[NomeMenu,PrologCmd],_).
```

Adicionado o predicado ao ficheiro *root.pl* o utilizador do CxIDE pode desfrutar do uso do novo predicado.

Exemplo de utilização:

```
%Adicionar dinamicamente um novo menu ao Eclipse que aquando da interação
%revelará na consola do CxIDE o estado atual de todas as Flags do CxProlog.
create_menu("Flags", "flags.")
```

8.3.3 Predicados que operam sobre menus

Foram desenvolvidos outros predicados envolvendo menus no Eclipse. Todos esses predicados têm uma implementação semelhada à revelada anteriormente.

Vejamos todos os predicados disponíveis aos utilizadores de CxIDE que operam sobre menus.

- **create_menu(NomeMenu)**

  Criar menu com o título "NomeMenu"

- **create_menu(NomeMenu,PrologCmd)**

  Criar menu com o título "NomeMenu" que executa "PrologCmd"

- **create_menu(NomeMenu,Item_Name,PrologCmd)**

  Criar menu com o título "NomeMenu" com um único item (submenu) com nome "Item_Name" e que executa a ação "PrologCmd"

- **create_menu(NomeMenu,SubMenus)**

  Criar menu com o título "NomeMenu" com submenus e respetivos comportamentos definidos em “SubMenus”

- **delete_menu(Menu_ID)**

  Elimina o menu com o id "Menu_ID",e se for um menu criado pelo utilizador todos os seus submenus também são eliminados. Caso contrário os submenus continuarão a existir

O predicado mais “poderoso” é o **create_menu**(NomeMenu,SubMenus), este possibilita a criação dum novo menu com vários submenus e seus respetivos comportamentos.

A variável **SubMenus** é uma lista que representa todos os submenus do novo menu a criar.

No entanto a lista **SubMenus** é organizada por pares, para cada submenu que se pretenda criar é necessário colocar o seu nome na lista e na posição seguinte o seu comportamento.

Exemplo de utilização:

```
create_menu('Cx File',['Open','file_chooser(Path),open_file(Path)',
  'Consult','editor_getCurrOpenFilePath(Path),consult(Path)']).
```

O exemplo acima cria o menu **Cx File** (figura 38) com dois submenus:

- **Open** - Definido por **file_chooser(Path)** e **open_file(Path)** (anexo 3), basicamente abre um diálogo para pesquisar um determinado ficheiro. O ficheiro selecionado é aberto no editor do CxIDE.

- **Consult** - Definido por **editor_getCurrOpenFilePath(Path)**, **consult(Path)** que consulta o atual ficheiro em edição no CxIDE.

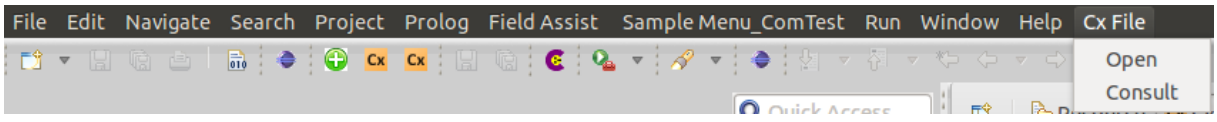


Figura 38 – Exemplo de adição dum menu ao Eclipse

Outro predicado interessante é o **delete_menu(Menu_ID)**. É possível com este predicado eliminar os menus criados pelo **create_menu** e até menus já existentes no Eclipse como o **File**. Só é assegurada a remoção dos submenus caso os menus envolvidos tenham sido criados através do CxProlog. Não foi encontrada nenhuma forma de garantir que por exemplo ao eliminar o menu **File** ou **Edit** que todos os seus submenus tivessem o mesmo destino.

Antes de terminar esta secção sobre menus é importante realçar um aspecto. O utilizador pode em determinada altura pretender aceder a uns dos menus que criou para alterar ou eliminar o mesmo. Acontece que o Eclipse não facilita esta tarefa, pois para aceder a um menu apenas disponibiliza o seguinte método **find** (figura 39).

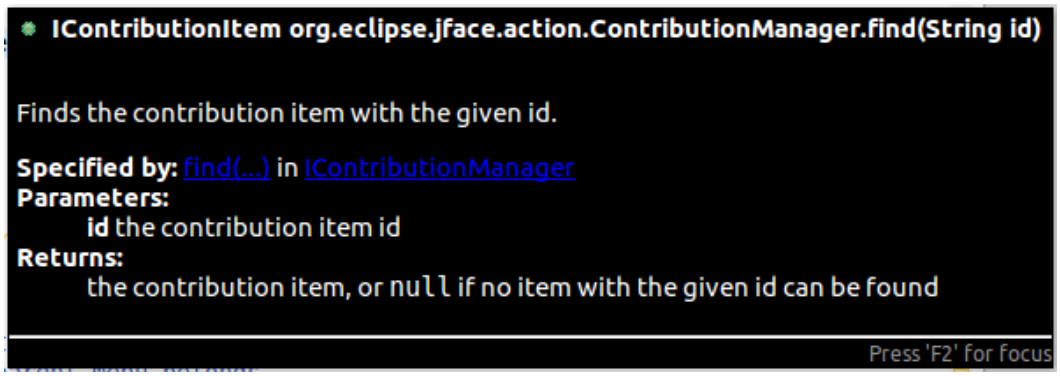


Figura 39 – Método find que opera sobre menus

O problema deste método é a necessidade de identificar o menu pai do menu que se quer encontrar. Ou seja, não existe um acesso direto a um menu apenas usando por exemplo o seu ID.

O CxIDE pretende que as suas extensões sejam fáceis de utilizar e portanto aproveitou a oportunidade para melhorar uma falha do Eclipse.

O objetivo era conseguir um acesso direto e fácil a qualquer dos menus criados pelo utilizador do CxIDE. Para isso foi criada uma estrutura de dados para armazenar todos os menus criados no CxIDE, mais concretamente um dicionário. Esse dicionário, dado o nome do menu, permite um acesso direto ao mesmo. Cada vez que o utilizador cria um menu, o mesmo, é adicionado ao dicionário. Este estilo de arquitetura permitiu implementar o predicado **delete_menu** que apenas necessita do nome do menu alvo. Quando o utilizador executa o **delete_menu** o dicionário interno do CxIDE é utilizado para rapidamente eliminar o menu em causa.

De forma a distinguir menus criados no CxIDE de menus já existentes no Eclipse (instâncias de **MenuManager**), todos os menus criados no CxIDE são **CxMenu**.

```
public class CxMenu extends MenuManager
```

Esta distinção é importante pois possibilita tratamento específico para os **CxMenus**. Por exemplo, é possível só ter o **CxMenus** visíveis quando a perspetiva do CxIDE está ativa, quando o utilizador mudar para a perspetiva de desenvolvimento Java os **CxMenu** não estarão visíveis.

8.3.4 Janelas de diálogo

As janelas de diálogo permitem a inserção de dados e a acção executada em princípio estará relacionado com os dados inseridos.

Assim foi decidido permitir ao utilizador de CxIDE a criação de diálogos usando CxProlog. Vejamos como os mesmos foram implementados, voltando novamente a seguir a receita (ver 8.3).

1. **Desenvolver o lado Java da extensão programaticamente.**

Para criar programaticamente diálogos em Java/Eclipse é necessário primeiro criar uma classe que estenda a classe **Dialog** disponibilizada pelo SWT.

Assim, foi criada a classe CxDialog sendo apresentado parte do seu código seguidamente.

```
public class CxDialog extends Dialog {
    private String message;
    private static IWorkbenchWindow window
    Workbench.getInstance().getActiveWorkbenchWindow();
    private static Shell shell = window.getShell();
```

```
//campos para inserção de texto
private LinkedHashMap<String, String> forms;
//campos para inserção de númericos
private LinkedHashMap<String, Double> numForms;
//ação a executar
private static Action action = null;
 //Cria um novo diálogo com um dado título
 //Com um estilo predefinido
public CxDialog(String title) {
      this(shell, title, SWT.DIALOG_TRIM | SWT.APPLICATION_MODAL);
}
//Cria um novo diálogo com um dado título e estilo
public CxDialog(Shell parent, String title, int style) {
      super(parent, style);
      forms = new LinkedHashMap<String, String>();
      numForms = new LinkedHashMap<String, Double>();
      setText(title);
}
//Revela o diálogo ao utilizador
public void open() {
      // Create the dialog window
      Shell shell = new Shell(getParent(), getStyle());
      shell.setText(getText());
      createContents(shell);
      shell.pack();
      shell.open();
      Display display = getParent().getDisplay();
      while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) {
                  display.sleep();
            } } }
```

Esta classe possibilita ao utilizador de CxIDE a criação de diálogos durante a execução do CxIDE utilizando o Java. Os diálogos podem ser simples contendo apenas um título e dois botões: **OK** para efetuar o comportame0nto definido, e **CANCEL** para não realizar o comportamento.

Obviamente o poder dos diálogos é de dar a possibilidade ao utilizador de inserir dados e que esses mesmos dados sejam usados pela ação do diálogo.

Era pretendido oferecer ao utilizador do CxIDE uma forma fácil de criar e modificar diálogos. E modificar, neste caso, significa: adicionar campos de inserção de texto, adicionar campos de inserção de numéricos e a própria definição do comportamento do diálogo.

2. **Testar se a extensão pode ser realizada durante a execução do Eclipse.**

O código acima apresentado conseguiu criar, sem problemas, simples diálogos durante a execução do Eclipse.

No entanto, o objetivo final era que todo o processo de criação de diálogos e sua modificação fosse realizada através do CxProlog, assim continuemos a receita.

3. **Desenvolver um método estático em Java de interface com o Prolog.**

No anexo 4 o leitor poderá encontrar o código Java relacionado com o **3º** passo acima (devido à sua dimensão foi anexado).

4. **Criar um predicado CxProlog que utilize java_call/2 para executar o método desenvolvido no passo 3.**

Com o método Java que cria um diálogo concluído era agora possível criar uma cláusula CxProlog que invocaria o anterior método, dessa forma seria possível criar um diálogo no Eclipse através do CxProlog. Eis a cláusula criada:

```
create_dialog(Titulo):-
java_call('org/eclipse/cxide/Menu_ops/Dialog_Operations','createDialog:(Ljava/lang/String;)V',[Titulo],_).
```

5. **Testar.**

O teste à execução, no Eclipse, dos métodos estáticos relacionados com os diálogos foi um sucesso. Não existiu qualquer problema em criar e alterar diálogos durante a execução do Eclipse. Na próxima secção é revelado um exemplo da criação de um diálogo.

8.3.5 Predicados relacionados com diálogos

Com os métodos estáticos implementados (anexo 4) foi possível implementar novos predicados CxProlog que operavam sobre diálogos no Eclipse.

Eis todos os predicados disponibilizados pelo CxIDE para a implementação de diálogos:

- **create_dialog(Titulo)**
  Criar diálogos
- **add_textForm_dialog(Titulo, Label)**
  Adicionar campos de inserção de texto.
- **add_numForm_dialog(Titulo, Label)**
  Adicionar campos de inserção de numéricos.
- **getTextField(Dialog, Field_Label, Value)**
  Obter o valor de campos de texto.
- **getNumField(Dialog, Field_Label, N)**
  Obter o valor de campos numéricos.
- **dialog_setAction(Dialog, Action)**
  Modificar o comportamento do diálogo.
- **dialog_setOutput(Dialog, Lista)**
  Cria um diálogo que exibe os elementos da Lista, esta cláusula é utilizada para fornecer um destacado output visual sempre que desejado.

Seguidamente, um exemplo da criação dum diálogo que aplica a fórmula resolvente a três valores:

```
 %Predicados para a fórmula resolvente
delta(A, B, C, D):- D is B*B - 4*A*C.
equation(A,B,C,R1,R2):-delta(A,B,C,D),R1 is (-B+sqrt(D))/(2*A), R2 is (-B-sqrt(D))/(2*A).

% Criação de diálogo sobre a Fórmula resolvente
:-create_dialog('Fórmula Resolvente').
:-add_numForm_dialog('Fórmula Resolvente','a').
:-add_numForm_dialog('Fórmula Resolvente','b').
:-add_numForm_dialog('Fórmula Resolvente','c').
:-dialog_setAction('Fórmula Resolvente','getNumField(''FórmulaResolvente'',''a'',A),getNumField(''Fórmula
Resolvente'',''b'',B),getNumField(''FórmulaResolvente'',''c'',C),equation(A,B,C,R1,R2), writeln(R1),writeln(R2)').
```

O código acima tem como resultado o seguinte diálogo:

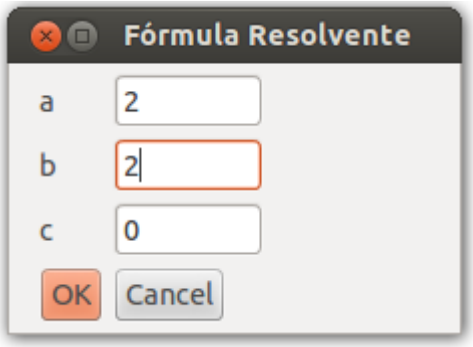


Figura 40 – Exemplo de diálogo criado via CxProlog

O resultado da execução do diálogo será apresentado na consola interna do CxIDE.

Tal como no caso dos menus foi necessário criar um dicionário (*String*, *CxDialog*) que associa o título dum diálogo à respetiva instância CxDialog. Cada diálogo tem uma lista de todos os seus campos de inserção (figura 41). Estas estruturas de dados permitem facilmente aceder e modificar durante a execução do CxIDE um diálogo criado pelo utilizador.

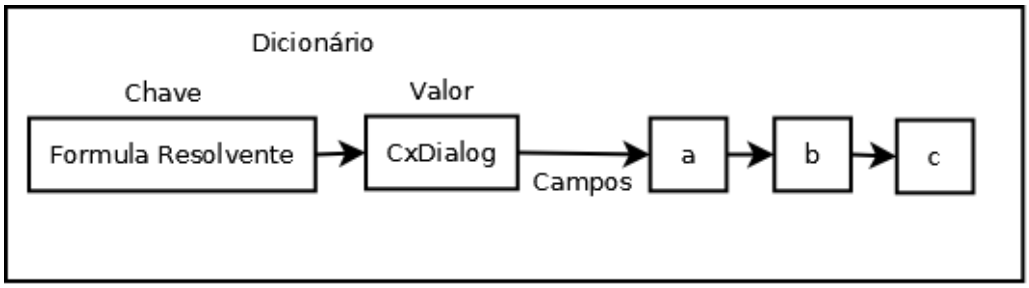


Figura 41 – Estruturas de dados relacionadas com diálogos

8.4 Editor

Para implementar um editor no Eclipse é necessário estender uma das várias classes disponibilizadas para esse efeito pela plataforma Eclipse (**TextEditor**, **MultiPageEditor**, etc.).

O CxEditor foi criado como subclasse da classe **TextEditor**[23].

```
public class CxEditor extends TextEditor
```

Assim definido, sem acrescentar mais nada, o **CxEditor** implementa um editor de texto muito básico. Facilitando ao utilizador algumas operações como **copy**, **paste** ou **find** que herdou do **TextEditor**.

O **CxEditor** tem uma identidade própria sendo possível distingui-lo de outros editores do Eclipse. O anterior facto possibilita, por exemplo, associar uma determinada extensão (ex: *.pl*) ao **CxEditor.** Quando o utilizador selecionar um ficheiro no Eclipse, com a extensão indicada, esse ficheiro será aberto automaticamente no **CxEditor**.

É pretendido que o **CxEditor** seja muito mais que um editor de texto básico (secção 6.4), e por isso é necessário adicionar algumas funcionalidades ao mesmo para benefício do utilizador do CxIDE.

[23] Mais informações sobre a classe TextEditor: goo.gl/BjS5yA

Nas secções seguintes é realizada a descrição da implementação de todas funcionalidades que o CxEditor disponibiliza. Serão igualmente introduzidos alguns conceitos relativos à plataforma Eclipse para facilitar ao leitor a compreensão do processo de implementação.

8.4.1 Realce de sintaxe

Quando o utilizador modifica o texto num editor do Eclipse, surge a necessidade de reexibir o texto para mostrar as mudanças realizadas.

O Eclipse utiliza o modelo *damage, repair, reconcile* para lidar com mudanças ao texto em edição (29).

No modelo *damage/repair/reconcile*, temos:

- **Damage** – Identificação do texto que deve ser reexibido.
- **Repair** – Extração de toda a informação necessária do texto *danificado* de forma a descrever todas as reparações necessárias.
- **Reconcile** – Manutenção da representação visual do documento.

Para definir o modelo *damage/repair/reconcile* a utilizar no **CxEditor** foi necessário instanciar a classe da plataforma Eclipse **SourceViewerConfiguration**[24]. Trata-se duma classe central para a configuração do editor, possibilitando a adição de funcionalidades como o realce de sintaxe e completação automática. Por isso, foi criada a subclasse **CxSourceConfig** que foi usada ao longo do projeto para adição de novas funcionalidades ao editor.

```
public class CxSourceConfig extends SourceViewerConfiguration
```

No realce de sintaxe é necessário identificar certos padrões que representam sequências de texto que necessitam dum tratamento próprio (ex: comentários começam com %). Por outras palavras, é necessário realizar análise lexical.

A classe **RuleBasedScanner**[25] da plataforma Java é usada para definir as regras que identificam os diferentes termos que são utilizados no realce de sintaxe (figura 42).

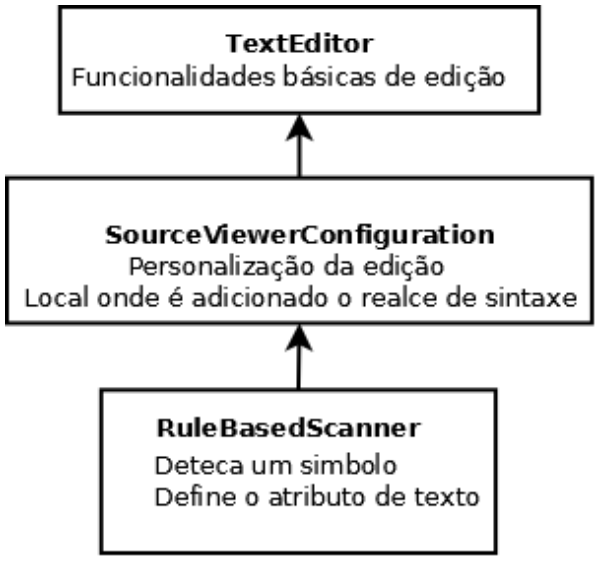


Figura 42 – Relações das classes relacionadas com o realce de sintaxe

Foi necessário criar a classe **CxScanner** que estende a classe **RuleBasedScanner**, com o objetivo de definir as regras do realce de sintaxe para o **CxEditor**.

[24] Mais informações sobre a classe SourceViewerConfiguration: goo.gl/1lxcef
[25] Mais informações sobre a classe RuleBasedScanner: goo.gl/hmkNLN

```
public class CxScanner extends RuleBasedScanner
```

Cada tipo de regra, que identifica um tipo de *token*, é descrito por uma subclasse da classe **IRule**[26] da plataforma Eclipse.

O **CxScanner** necessita duma enumeração das regras do realce de sintaxe, que é registada num vetor.

O Eclipse já fornece algumas regras, sobre a forma de classes, para a definição do realce de sintaxe. Seguidamente são reveladas as regras que foram utilizadas para implementar o realce de sintaxe do **CxEditor**:

```
SingleLineRule(String startSequence, String endSequence, IToken token,
                                        char escCharacter)
```

Esta classe, descreve linhas que se iniciam com determinada sequência de carateres e finalizam com outra sequência.

Exemplo de utilização:

Numa linha, texto entre plicas representa uma String e deve ser colorado de azul.

```
IToken stringToken = tokenFor(new RGB(0,0,255));
new SingleLineRule("'", "'", stringToken, '\\')
```

```
EndOfLineRule(String startSequence, Itoken token)
```

Define uma regra para a dada sequência inicial numa única linha, que se detetada retorna o *token* pretendido.

Exemplo de utilização:

Linhas que iniciem com o caráter % representam comentários e devem ser coloradas de vermelho.

```
IToken commentToken = tokenFor(new RGB(255,0,0));
addRule(new EndOfLineRule("%", commentToken));
```

```
MultiLineRule(String startSequence, String endSequence, IToken token)
```

Define uma regra para a dada sequência inicial e final, que se detetada retorna o *token* pretendido.

Exemplo de utilização:

Texto limitado com os carateres ** representa comentários e deverá ser colorado de vermelho.

```
IToken multiCommentToken = tokenFor(new RGB(255,0,0));
addRule(new MultiLineRule("**", "**", multiCommentToken));
```

```
WordRule(IWordDetector detector, Itoken defaultToken)
```

Cria uma regra para uma dada palavra, que se detetada retorna o *token* pretendido.

No entanto, é necessário definir um **WordDetector**[27] para clarificar o que é uma palavra.

Foi criado um **WordDetector** semelhante ao utilizado no IDE para Java do Eclipse.

Seguidamente é apresentado o código do WordDetector criado.

---

[26] Mais informações sobre a interface IRule: goo.gl/be22qF
[27] Mais informações sobre a interface IwordDetector: goo.gl/EMjlp6

```
public class WordDetector implements IWordDetector {
    /** Um caráter é o início de uma palavra se uma das seguintes condições for válida:
     o caráter for uma letra (isLetter)
     o caráter for um símbolo como '$'
     o caráter for usando como um caráter de conexão '_' **/
    @Override
    public boolean isWordStart(char c) {
        return Character.isJavaIdentifierStart(c);
    }
    //Um caráter é parte duma palavra se alguma das condições acima reveladas for verdade
    // mas agora pode ser também um número
    @Override
    public boolean isWordPart(char c) {
        return Character.isJavaIdentifierPart(c);}}
```

Criado o **WordDetector**, é possível adicionar palavras ao mesmo e o *token* que as representa. Assim quando essa palavra específica for detetada será realçada de acordo. É assim que o IDE para Java do Eclipse define a coloração de palavras como: "new", "public", "if".

Exemplo de utilização:

Definir a cor do texto por omissão para preto e a palavra "cxprolog" deverá ter a coloração azul.

```
WordDetector wordDetect = new WordDetector();
//cor do texto "normal"
IToken defaultToken = tokenForColor(new RGB(255,255,255));
WordRule wordRule = new WordRule(wordDetect, defaultToken);

IToken cxToken = tokenForColor(new RGB(0,0,255));
//adicionar uma nova regra para a coloração da palavra cxprolog
wordRule.addWord("cxprolog", defaultToken);
addRule(wordRule);
```

O **WordDetector** permite identificar palavras concretas, mas por vezes é necessário definir regras mais gerais (definidas por padrões). No caso do CxProlog seria importante dar uma distinção visual às variáveis.

O Eclipse oferece a interface **IPredicateRule**[28] para construir regras que identificam sequências de carateres que verificam determinados padrões.

Para identificar variáveis do CxProlog no **CxEditor** foi criada a seguinte classe:

```
public class VarRule implements IPredicateRule
```

A parte mais interessante desta classe é a que declara a avaliação realizada a uma sequência de carateres:

```
// Uma sequência de carateres é uma variável se for iniciada por underscore
//ou maiúsculas e terminar com um espaço branco
  public IToken evaluate(ICharacterScanner scanner, boolean resume) {
```

[28] Mais informações sobre a interface IpredicateRule em: goo.gl/G0x3tE

```
            scanner.unread();
            if (!wsdetector.isWhitespace((char)scanner.read()))
                  return Token.UNDEFINED;
            int c = scanner.read();
            if (!(c  == '_' || c >= 'A' && c <= 'Z') || c ==
ICharacterScanner.EOF) {
                  scanner.unread();
                  return Token.UNDEFINED;
            }
            do {
              c = scanner.read();
            } while (!wsdetector.isWhitespace((char)c) && c !=
ICharacterScanner.EOF);
            scanner.unread();
            return getSuccessToken();}
```

Exemplo de utilização:

As variáveis CxProlog devem ser coloradas de amarelo.

IToken variableToken = *tokenFor*(new RGB(255,255,0));

***addRule*(new VarRule(variableToken));**

Com as regras apresentadas anteriormente já era possível definir o realce de sintaxe que o **CxEditor** utilizaria.

Juntemos agora todos os exemplos anteriores, relativos às regras de realce de sintaxe, para visualizar o efeito que as mesmas teriam sobre um pequeno texto.

Regras	Exemplo de implementação

Comentários	IToken commentToken = *tokenFor*(new RGB(255,0,0));
	addRule(new EndOfLineRule("%", commentToken));
	
	IToken multiCommentToken = *tokenFor*(new RGB(255,0,0));
	addRule(new MultiLineRule("**", "**", multiCommentToken));

Strings	IToken stringToken = *tokenFor*(*new RGB(0,0,255)*); new SingleLineRule("'", "'", stringToken, '\\')
Variáveis	IToken variableToken = *tokenFor*(new RGB(255,255,0)); *addRule*(new VarRule(variableToken));

Texto padrão e palavras específicas	WordDetector wordDetect = **new** WordDetector();
	//cor do texto "normal"
	IToken defaultToken = *tokenForColor*(**new** RGB(255,255,255));
	WordRule wordRule = **new** WordRule(wordDetect, defaultToken);
	
	IToken cxToken = *tokenForColor*(**new** RGB(0,0,255));
	//adicionar uma nova regra para a coloração da palavra cxprolog

	wordRule.addWord("cxprolog", defaultToken); ***addRule*(wordRule);**

Tabela 2 - Exemplos de definições de regras de realce de sintaxe

Ficheiro sem realce	Ficheiro com o realce acima definido
% Um comentário de uma linha **** Comentários** **de várias** **linhas **** **cxprolog('realce de sintaxe').** **vars(A , _B, Ccc).** **textoNormal.**	**% Um comentário de uma linha** **** Comentários** **de várias** **linhas **** **cxprolog('realce de sintaxe').** **vars(A , _B, Ccc).** **textoNormal.**

Tabela 3 – Exemplo de aplicação de regras de realce de sintaxe

O realce de sintaxe até agora desenvolvido já revelava interessantes destaques visuais que beneficiariam o utilizador do CxIDE, mas faltava o tratamento dos termos específicos da linguagem.

No caso do CxIDE foi decidido adicionar realce sintaxe suplementar apenas para os nomes dos predicados builtin.

Para realizar o realce de tais nomes poderia usar-se a **WordRule** adicionando um a um todos os nomes dos predicados *builtin* do CxProlog, atribuindo-lhes o mesmo realce.

A dificuldade que surgiu neste ponto foi como obter esses nomes. Uma solução seria ler o manual CxProlog e colocar *hardcoded*, um a um, todos os nomes dos predicados *builtin* encontrados. Mas esta solução seria algo penosa e demorada. O mais preocupante é que o código não ficaria de todo extensível ou genérico. Por isso, foi ponderada uma outra solução.

Já que o CxIDE utiliza a biblioteca dinâmica do CxProlog, porque não obter dessa biblioteca uma lista com nome de todos os predicados *builtin*. Em Prolog é fácil criar predicados que operam sobre listas e o melhor é que o CxProlog oferece o seguinte predicado que permite obter predicados com determinadas propriedades:

```
predicate_property(?Head,?Prop)
```

Head identifica um predicado com a propriedade **Prop**.
Sendo possível obter todos os predicados com determinada propriedade através do *backtraking*.

A propriedade pretendida é a *built_in*. É necessário *backtraking* para obter todos os predicados com certa propriedade utilizando o **predicate_property**. Por isso, foi necessário desenvolver um novo predicado para facilitar a obtenção duma lista com todos os nomes dos predicados *builtin*.

Eis os predicados criados:

```
%Obtém a lista de predicados com dada propriedade
getPredicates(Prop,List):findall(Pred,predicate_property(Pred,Prop),List)
.
```

```
   %Recebe uma lista de predicados e retorna a lista com os seus nomes
   getFunctors([],[]).
   getFunctors([X|Xs],[Fun|Ys]) :- functor(X, Fun, Ar), getFunctors(Xs,Ys).
   %Obtém a lista de nomes de todos os predicados com a propriedade Pro
   getPredicatesWithProperty(Prop,Pre):-
getPredicates(Prop,L),getFunctors(L,Pre).
```

Exemplo de utilização:

```
[main] ?- getPredicatesWithProperty('built_in',Pre).
 Pre = [pwd,pop,push,zzz,foreigns,gui_text_close,... ]
```

O problema agora é que estas informações estavam apenas disponível do lado do CxProlog.

Era necessário conseguir passar esta lista de nomes para o Java (secção 6.6), para que a mesma fosse utilizada nas regras do realce de sintaxe.

Para esse efeito, foi desenvolvido um método estático no Java que receberia um vetor de *strings* e o guardaria numa variável estática (String []) denominada **builtins**. O **builtins** é um vetor de *strings* que contém todos os nomes dos predicados com a propriedade "built_in" do CxProlog.

Este vetor poderia ser utilizado para adicionar todos os seus elementos ao **WordRule** para definir o realce de sintaxe dos predicados *builtin* do CxProlog no **CxEditor**.

```
//Método estático para definir o vetor com nomes de builtins CxProlog
   public static void setBuiltins(String[] b){
            builtins=b;
      }
```

Falta apenas desenvolver um predicado com o *java_call/2* que utilize o método estático **setBuiltins** passando-lhe a lista obtida através do **getPredicatesWithProperty('built_in',Pre)**.

```
getBuiltins(Lista):-
getPredicatesWithProperty('built_in',Lista),writeln(Lista),java_call('org/e
clipse/cxide/utilities/Editor_Utilities','setBuiltins:([Ljava/lang/String;)
V',[Lista],_).
```

O Java podia agora utilizar o **CallProlog** para executar o **getBuiltins** (acima) obtendo assim a lista dos nomes dos predicados *builtin* CxProlog. Sendo apenas necessário percorrer a lista obtida adicionando cada um dos elementos no **WordRule** (figura 43).

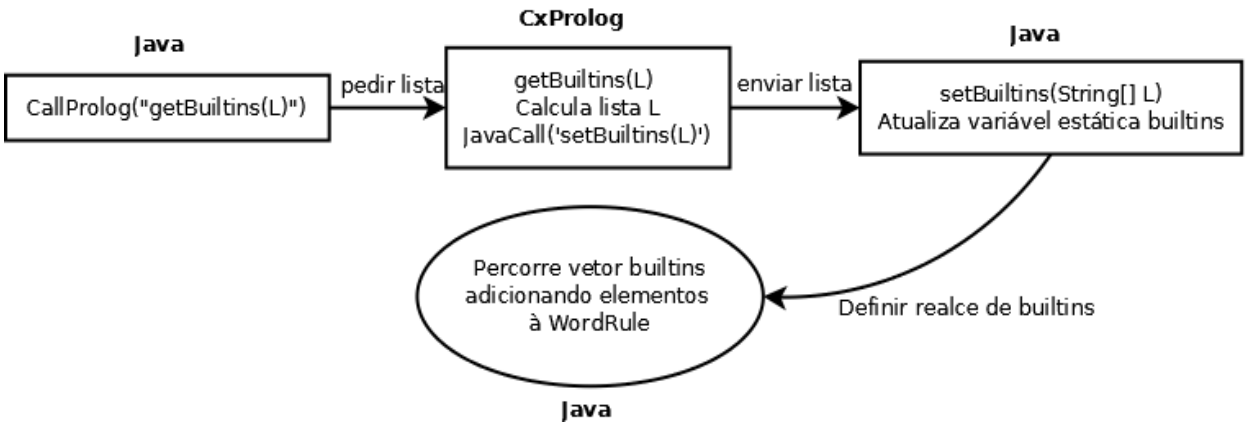


Figura 43 – Implementação do realce de sintaxe de predicados builtin

Podia dar-se como concluído o realce de sintaxe do **CxEditor**, mas uma das ambições do projeto é que o utilizador consiga configurar o CxIDE utilizando o CxProlog. E porque não oferecer ao utilizador do CxIDE a oportunidade de definir as suas próprias regras de realce de sintaxe usando o CxProlog... Já era sabido como definir programaticamente em Java o realce de sintaxe. Faltava assim criar métodos estáticos que fizessem o mesmo e seguidamente criar predicados CxProlog que usem esses métodos através do **java_call/2**.

Foram criados os cinco seguintes métodos estáticos para definir o realce de sintaxe:

```
  public static void singleLineRule(String fieldName, String startSeq,
String endSeq, String escChar, int red, int green, int blue)
  public static void endOfLineRule(String fieldName, String startSeq, int
red, int green, int blue)
  public static void addDefaultTextRule(String fieldName, int red, int
green, int blue)
  public static void addVarRule(String fieldName, int red, int green, int
blue)
  public static void addWordRule(String fieldName,String word, int red, int
green, int blue)
```

Existem os cinco seguintes predicados CxProlog para definir o realce de sintaxe do CxIDE:

```
editor_singleLineRule(FieldName, StartSeq, EndSeq, EscChar, R, G, B)
editor_endLineRule(FieldName,StartSeq,R,G,B)
editor_varRule(FieldName,R,G,B)
editor_addNormalTextRule(FieldName,R,G,B)
editor_addWordRule(FieldName,Word,R,G,B)
editor_addWordsRule(FieldName,[Head|Tail],R, G, B)
```

A variável **FieldName** é usada para identificar a regra em causa. Esta identificação é utilizada nas preferências do CxEditor. Ou seja, não só o utilizador do CxIDE pode configurar o realce de sintaxe, como para cada regra que crie é automaticamente gerado um novo campo nas preferências do CxIDE. Assim o CxIDE oferece, tal como outros IDEs para Eclipse, a hipótese de facilmente modificar a coloração de determinados termos através do menu preferências do seu Editor.

O leitor interessado poderá consultar o anexo 2, para obter mais informações sobre como definir o realce de sintaxe do **CxEditor**.

8.4.2 Preferências do editor

Na secção anterior já foi apresentada uma parte das preferências do CxIDE, mas a sua implementação ainda não foi revelada.

Uma página de preferências é uma janela de diálogo que é acessível através do menu de preferências do Eclipse.

O Eclipse disponibiliza a extensão **org.eclipse.ui.preferencePages**[29] para facilitar a criação de preferências. Esta extensão permite a adição de novas páginas de preferências, isto é, novas entradas no menu do Eclipse:

Window → Preferences

Para criar um menu com o nome "CxProlog" e uma página de preferências foi criada a classe CxPrologPreferences.

[29] Mais informações sobre a extensão org.eclipse.ui.preferencePages em: goo.gl/EyuzT0

A CxPrologPreferences é uma subclasse da classe **FieldEditorPreferencePage**[30] e que implemente a interface **IWorkbenchPreferencePage**[31]**.**

```
public class CxPrologPreferences extends FieldEditorPreferencePage
                implements IWorkbenchPreferencePage
```

FieldEditorPreferencePage – Possibilita a definição de páginas de preferências com campos de inserção de dados.

IWorkbenchPreferencePage – Possibilita a definição de um novo menu nas preferências do Eclipse.

A parte mais interessante da classe CxPrologPreferences é aquela que possibilita a adição de novos campos para a inserção de dados à página.

```
public void createFieldEditors() {
//Cria um campo que receberá o caminho dum ficheiro
addField(new FileFieldEditor(PreferenceConstants.CX_PATH,
          "CxProlog Path:", getFieldEditorParent()));
//Cria um campo que receberá uma String
addField(new StringFieldEditor(PreferenceConstants.SV_IP, "Server Ip:",
          getFieldEditorParent()));
//Cria um campo que receberá um inteiro
addField(new IntegerFieldEditor(PreferenceConstants.SV_PORT,
          "Server Port:", getFieldEditorParent()));
}
```

Eis a página de preferências criada pelo código acima exibido:

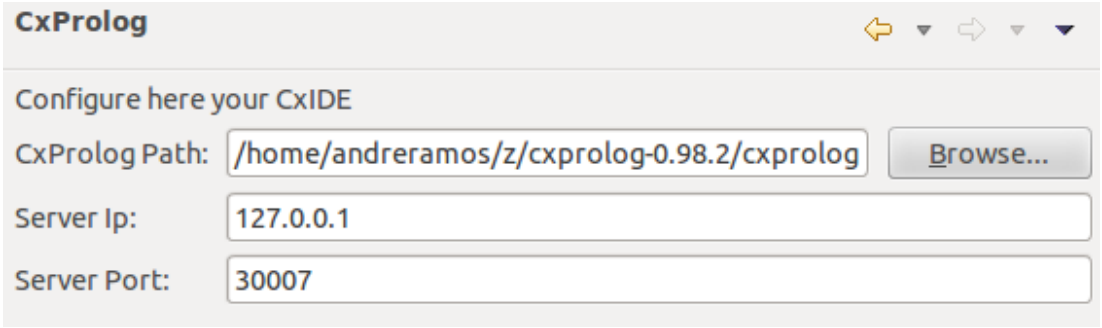


Figura 44 – Página de preferências do CxIDE

Os nomes das etiquetas dos campos (ex: "*CxProlog Path:*") são definidos na classe **PreferenceConstants**.

No CxIDE, o controlo das preferências faz-se através de duas páginas. A primeira já foi apresentada e a segunda está relacionada com o realce de sintaxe. Nessa mesma secção foi realçado o facto de o utilizador de CxIDE poder definir as preferências de realce, durante a execução do Eclipse, utilizando o CxProlog. A segunda página de preferências foi mais difícil implementar pois a mesma pode ser modificada durante a execução do Eclipse.

Algo interessante das páginas de preferência é que as mesmas são criadas de raiz cada vez que o utilizador abre a página. Ou seja, todo o código envolvido na criação das páginas é executado cada vez

[30] Mais informações sobre a classe FieldEditorPreferencePage em: goo.gl/UOQUqh
[31] Mais informações sobre a interface IWorkBenchPreferencePage em: goo.gl/Qa5bwe

que o utilizador abre essa página. O anterior facto possibilita a modificação de tais páginas dinamicamente.

A página de preferências do realce de sintaxe está dependente duma estrutura de dados, para saber que campos deverá incluir na sua visualização. Cada vez que o utilizador define uma regra de realce de sintaxe são adicionadas informações à estrutura de dados. Aquando da abertura da página de preferência a estrutura de dados é consultada atualizando a página de acordo.

A estrutura de dados em causa é um HashMap que associa uma String a uma cor(RGB):

```
private static HashMap<String,RGB> prefFields = new
HashMap<String,RGB>();
```

A String envolvida representa o nome do campo (*label*) da preferência. A cor representa a coloração usada no *token* respetivo.

Um exemplo tornará mais fácil a compreensão:

Considere-se a execução dos seguintes comandos:

```
:-editor_singleLineRule('word "','"', '"', '\\', 100, 100, 100).
:-editor_singleLineRule('word ''','''', '''', '\\', 50, 50, 50).
:-editor_endLineRule('comment %','%', 150, 0, 150).
:-editor_varRule('variables',0,255,0).
:-editor_addNormalTextRule('default text',0,0,0).
:-editor_addWordRule('special','andre',200,200,0).
:-editor_addPropertyHighlight('built_in',0,0,255).
:-editor_addPropertyHighlight('user_defined',0,255,0).
```

Com os comandos acima, o **HashMap** *prefFields* fica preenchido com os seguintes valores:

word “	RGB(100,100,100)
word '	RGB(50,50,50)
comment %	RGB(150,0,150)
variables	RGB(0,255,0)
default text	RGB(0,0,0)
special	RGB(200,200,0)
built_in	RGB(0,0,255)
user_defined	RGB(0,255,0)

Tabela 4 – Exemplo de instanciação do HashMap relativo ás preferências do realce de sintaxe

Sendo criada a seguinte página de preferências:

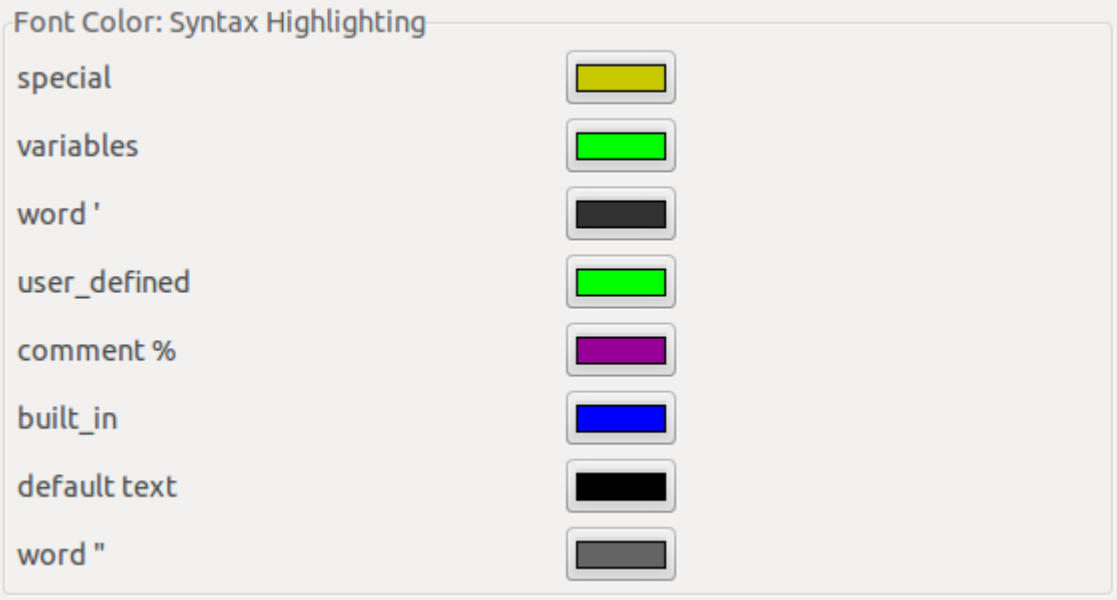


Figura 45- Preferências do CxEditor

Como é visível, os campos inicialmente já possuem um valor por omissão.

Para inicializar as preferências, o Eclipse disponibiliza a extensão:

org.eclipse.runtime.preferences

Esta extensão possibilita ao programador a inicialização das preferências. Para definir as inicializações foi criada a classe **PreferenceInitializer** que é subclasse de **AbstractPreferenceInitializer**[32].

```
public class PreferenceInitializer extends AbstractPreferenceInitializer
```

Cujo seu único método é:

```
//Atribuição de um valor inicial as preferências
public void initializeDefaultPreferences() {
        IPreferenceStore store =
Activator.getDefault().getPreferenceStore();
        //Inicializar página do realce de sintaxe
        HashMap<String, RGB> prefsFields =
EditorPreferences.getPrefFields();
        Iterator<String> it = prefsFields.keySet().iterator();
        //Percorrer o prefFields e definir os novos valores pagina
        while(it.hasNext()){
              String name = it.next();
              RGB color = prefsFields.get(name);
              PreferenceConverter.setDefault(store, name, color);
              store.setToDefault(name);
        }
        //Inicializar página do CxProlog
        store.setDefault(PreferenceConstants.P_BOOLEAN, true);
```

[32] Mais informações sobre a classe AbstractPreferenceInitializer em: goo.gl/xRJPoe

```
            store.setDefault(PreferenceConstants.SV_PORT,
PreferenceConstants.SV_PORT_VALUE);
            store.setDefault(PreferenceConstants.CX_PATH,
                        PreferenceConstants.CX_PATH);
            store.setDefault(PreferenceConstants.SV_IP,
                        PreferenceConstants.SV_IP_VALUE); }
```

O método acima utiliza o **setDefault(String name, Object object)** que procura o campo identificado por **name** e atribui-lhe o valor inicial **object**.

É aqui que entra em ação o **HashMap prefFields**. De cada vez que o utilizador abre a página das preferências relacionada com o **CxEditor** o código acima é executado. Ao executar o código acima é percorrido o *prefFields* atualizando a página das preferências consoante as regras definidas pelo utilizador.

Resumindo, cada vez que o utilizador adiciona uma nova regra de realce de sintaxe é criada uma respetiva entrada no *prefFields*. E cada vez que a página de preferências do **CxEditor** é aberta o *prefFields* é percorrido adicionando os campos e respetivos valores atualizados.

8.4.3 Completação automática

A completação automática é uma das principais funcionalidades que se pretende oferecer ao utilizador do CxIDE. A mesma apresenta várias vantagens durante a edição de ficheiros secção 4.1.4.

A classe **CxSourceConfig** é a responsável por instalar a completação automática no **CxEditor**. A definição da completação automática é realizada na classe CxPrologContentAssist que implemente a interface **IContentAssistProcessor**[33] da plataforma Eclipse:

```
public class CxPrologContentAssist implements IcontentAssistProcessor
```

A parte mais relevante desta classe situa-se no método:

```
public ICompletionProposal[] computeCompletionProposals(ITextViewer
                        viewer,int offset)
```

Neste método é acedido o atual ficheiro em edição no CxIDE (através do *viewer*), sendo obtida a sequência de carateres que o utilizador atualmente está a digitar. Essa sequência será comparada a uma outra sequência de carateres. Caso a primeira sequência (digitada pelo utilizador) seja um prefixo da segunda (armazenada internamente), é possível oferecer a segunda sequência como uma nova proposta de completação. Vejamos um exemplo:

Lista interna de possíveis completações		
André	Barbara	Cátia
Andreia	Beatriz	Claudio
Ana	Catarina	Celeste

Tabela 5 - Exemplo de completação automática, lista interna de completações.

[33] Mais informações sobre a interface IcontentAssistProcessor em: goo.gl/qbt2f9

Utilizador digita	Propostas de completação oferecidas
A	André, Andreia, Ana
And	André, Andreia
Ca	Catarina
Jos	
	André, Andreia, Ana, Barbara, Beatriz, Catarina, Cátia, Claudio, Celeste

Tabela 6 - Exemplo de completação automática, geração de propostas.

De realçar alguns casos interessantes que o exemplo revela:

- Caso o utilizador não digite nada, obterá todas as propostas de completação existentes.
- Caso o utilizador digite uma palavra que não seja prefixo de qualquer das possíveis propostas então o utilizador não obterá nenhuma proposta de completação.
- A acentuação é utilizada para distinguir. Quando o utilizador digitou "*Ca*" não obtém Cátia como proposta de completação.

Voltando ao método **computeCompletionProposals**, o mesmo tem de retornar um:

ICompletionProposal[]

ICompletionProposal[] é um vetor com as propostas de completação para algo que o utilizador digitou, no exemplo (tabela 6) seria as "Propostas de completação oferecidas". Mas para originar as tais propostas de completação é necessária uma lista interna com todas as possíveis completações. Foi decidido que o **CxEditor** ofereceria como propostas de completação todos os predicados *built_in* do CxProlog.

No realce de sintaxe foi necessário obter a lista de todos os nomes dos predicados *builtin* do CxProlog. Agora é necessário muito mais. A ambição é dar ao utilizador uma completação automática semelhante à oferecida pelo IDE Java do Eclipse (figura 46).

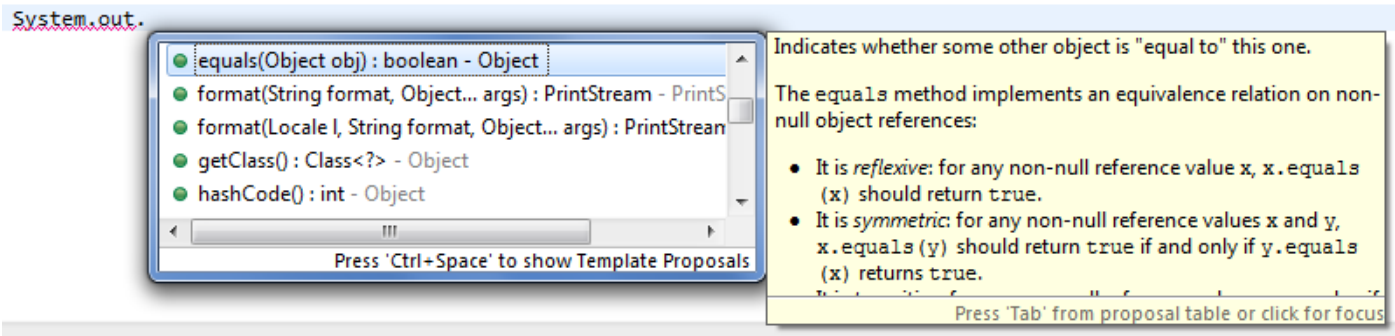


Figura 46 – Completação automática do editor de Java do Eclipse

Para obter tal completação automática é necessário ter acesso à assinatura de todos os predicados *builtin* do CxProlog, tal como às descrições de cada um.

O CxProlog não oferece estas informações, ou seja não existe nenhum predicado *builtin* que retorna as assinaturas ou descrições dos predicados existentes. No limite talvez fosse possível desenvolver um novo predicado que construi-se as assinaturas a partir do "funtor" e "aridade" dos predicados. Mas não seria de todo possível obter as suas descrições. O único sítio em que era possível obter as descrições dos predicados CxProlog era no seu manual (20).

A decisão mais fácil seria apenas oferecer uma completação automática no CxIDE com as assinaturas dos predicados e sem qualquer tipo de descrição do que esse predicado faria.

Uma das dificuldades ao desenvolver este projeto foi a necessidade de saber como determinado predicado CxProlog devia ser usado. Muitas vezes era necessário recorrer ao manual para tirar as dúvidas existentes. Não era de todo pretendido que o utilizador de CxIDE tivesse de enfrentar as mesmas dificuldades. Era necessário encontrar uma outra solução.

O manual do CxProlog foi desenvolvido como um **reStructuredText**[34], e resumidamente, o manual está atualmente num formato de texto (.txt) mas pode ser convertido automaticamente para outros formatos como HTML ou XML.

Era sabido que ao obter um formato XML do manual do CxProlog, era possível utilizar o Java para processar esse XML extraindo as informações desejadas.

O plano era então converter o manual CxProlog para um XML com o objetivo de aceder às assinaturas e descrições dos predicados que o mesmo oferece.

Rapidamente o plano foi posto em causa pois apesar da conversão automática do manual CxProlog para XML ter resultado, o ficheiro obtido apresentava muitas zonas corrompidas. Uma das causas era a acentuação, que aquando da conversão de TXT para XML parecia gerar problemas. Existiam outras zonas corrompidas que não envolviam acentuação em que não foi possível descobrir a origem dos problemas.

Ainda existia alguma esperança pois foi possível de facto utilizar o Java para processar as zonas não corrompidas do XML. Portanto, só faltava ter um XML sem qualquer problema para continuar a implementação da completação automática.

Como não foi possível uma conversão automática com sucesso, foi decidido realizar uma conversão manual. Seria criado um XML de raiz, baseado no manual do CxProlog, que contivesse todas as assinaturas de predicados e suas respetivas descrições.

Foi um processo bastante penoso e algo demorado, mas o XML final foi um sucesso.

Eis um excerto do manual do CxProlog em formato XML:

```
<pred>
    <id>push(+U)</id>
    <desc>push +U
    Pushes the unit designator U on top of current
    context. Therefore changes the current unit
    to be used in the top-level iteration.</desc>
</pred>
```

O XML resultado (*builts.xml*) era constituído por vários elementos **<pred>** que representam predicados. Cada elemento **<pred>** tem dois elementos :

<id> assinatura do predicado

[34] Mais informações sobre o reStructuredText: docutils.sourceforge.net/rst

<desc> descrição do predicado

Foram criadas duas classes envolvidas no processamento do XML:

```
public class ExtractBuiltins
```

```
public class PredInfo
```

A classe **ExtractBuiltins** é responsável por processar o *builts.xml* utilizando o **org.w3c.dom**[35].

No final do processamento estaria disponível um vetor de **PredInfo**

Um **PredInfo** representa um predicado e permite armazenar a sua assinatura e descrição.

Resumindo agora todo o funcionamento da completação automática do CxIDE.

O **CxSourceConfig** permite instalar no **CxEditor** a completação automática.

A classe **CxPrologContentAssis**t implementa a completação automática. É utilizado um documento XML com todas as informações do manual do CxProlog sobre predicados *builtin* para criar uma lista interna de propostas de completação. O utilizador ao digitar texto no CxIDE pode premir **CTRL+SPACE**. Nesse momento a sequência de carateres que o utilizador digitou será utilizada para verificar se a mesma é um prefixo de algumas propostas de completação. Se for um prefixo, as propostas de completação envolvidas serão reveladas ao utilizador (figura 47).

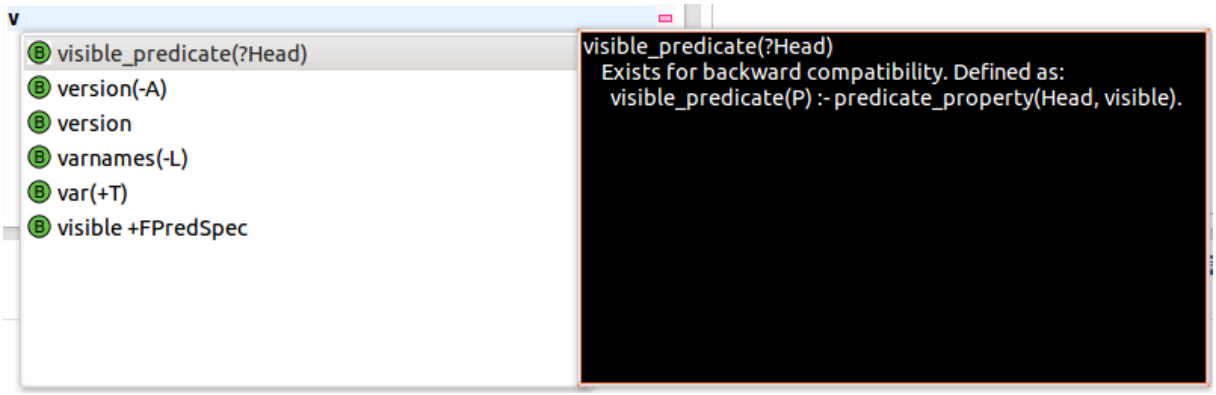


Figura 47 – Completação automática do CxEditor

8.4.4 Validação sintática

A possibilidade de validar o código em edição a qualquer altura é uma mais-valia para um IDE. O utilizador pode assim verificar facilmente se o seu atual projeto apresenta alguns erros de sintaxe antes de executar o mesmo.

IDEs como o PDT realizam a validação sintática através de um *parser* interno. Mais concretamente, o PDT implementou de raiz um *parser* em JavaCC[36] para analisar ficheiros usando as regras de sintaxe do Prolog padrão. Sendo esse *parser* utilizado para a validação sintática. No entanto, o PDT desfruta de uma ligação com o SWI-PROLOG e esse já tem o seu próprio *parser*.

Mas será de facto necessário implementar um parser em Java para obter validação sintática num editor do Eclipse?

A resposta é não, e uma solução é dada pelo CxIDE.

O CxIDE utiliza a biblioteca dinâmica do CxProlog e a mesma já tem um *parser* completamente dedicado a CxProlog. Foi decidido não implementar um *parser* de raiz mas sim utilizar o *parser* da biblioteca dinâmica no CxIDE.

Para implementar a validação sintática no CxIDE foi necessário:

---

[35] Mais informações sobre o org.w3c.dom em: goo.gl/XsDIfC
[36] Mais informações sobre JavaCC em: javacc.java.net

Criar o predicado **code_summary.** Este predicado permite extrair informação sintática dum ficheiro. Eis a assinatura e descrição do **code_summary**:

code_summary(Code,Summary,Errors)

Code – O texto a analisar

Summary – Uma lista com seis elementos por cada predicado encontrado no texto.

Esses seis elementos são o seguinte:

1º Posição medida em caracteres do inicio da cláusula.

2º Número da linha onde se inicia a cláusula.

3º Posição medida em caracteres dentro da linha do inicio da cláusula.

4º Posição medida em caracteres do final da cláusula.

5º Número da linha onde se finaliza a cláusula.

6º Posição medida em caracteres dentro da linha do final da cláusula.

Errors - Uma lista com sete elementos por cada erro encontrado no texto.

Os primeiros seis elementos delimitam a cláusula no texto, sendo uma repetição do caso anterior.

Há apenas um novo elemento:

7º Mensagem de erro.

Na validação sintática apenas é relevante a lista **Errors**, que identifica os erros sintáticos dum texto. A lista **Summary** fornece as localizações de todos os predicados dum texto sendo utilizada na implementação do colapso e expansão (secção 8.4.6) e da vista estruturada (secção 8.4.7). De realçar, que antes da implementação do predicado **code_summary** já se antevinha a necessidade pelas informações sobre o tipo de erros e a sua localização para o realce de erros, tal como a localização de predicados para o colapso e expansão. Dessa forma, o **code_summary** foi implementado para dar resposta a ambas as necessidades.

Eis um exemplo da utilização do **code_summary** num pequeno texto:

Texto a analisar	Lista Errors retornada pelo code_summary
ola(ana. ola(ana). ola(a na)	[1, 1, 1, 8, 2, -1, % SYNTAX ERROR: expected ')'. ola(ana#HERE#., 22, 5, 1, 31, 8, -1, % SYNTAX ERROR: Premature end of file. ola(a na)#HERE#]

Tabela 7 – Informações do CxProlog relacionados com a análise sintática

Existem várias informações sobre a localização dos erros para fins de realce dos mesmos. O realce de erros será descrito na secção seguinte.

Faltava agora conseguir obter o texto em edição no CxEditor para utilizar com o **code_summary**.

Para, programaticamente em Java, obter o conteúdo do editor ativo é usado o seguinte código:

```
public static String getCurrentEditorContent() {
    final IEditorPart editor =
    PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage()
    .getActiveEditor();
    if (!(editor instanceof ITextEditor)) return null;
    ITextEditor ite = (ITextEditor)editor;
    IDocument doc =
    ite.getDocumentProvider().getDocument(ite.getEditorInput());
    return doc.get();}
```

Com o método estático Java implementado é possível criar um predicado CxProlog que execute o mesmo.

O predicado seguinte permite que o CxProlog tenha acesso ao atual conteúdo do **CxEditor**:

```
editor_getCurrentContent(Content):-
java_call('org/eclipse/cxide/Menu_ops/Editor_Operations','getCurrentEditorC
ontent:()Ljava/lang/String;',[], Content).
```

Com os predicados **code_summary** e **editor_getCurrentContent** já era possível validar o texto em edição do **CxEditor**.

Por exemplo, quando o utilizador guarda o ficheiro em edição (*save*), poderia ser executada a seguinte chamada ao CxProlog para validar o atual ficheiro em edição.

```
Prolog.CallProlog("editor_getCurrentContent(Content),code_summary(Content
                                ,_,Errors)")
```

De facto com o código acima já se poderia avisar o leitor se existiriam erros ou não. No entanto o *feedback* seria dado pela consola.

Modernamente, os programadores esperam ver o resultado duma validação sintática com um realce de erros, o característico sublinhado vermelho.

8.4.5 Realce de erros

O leitor atento terá evidenciado que o **code_summary** não se limita a identificar os erros existentes. Várias informações sobre a localização dos erros são disponibilizadas pelo **code_summary**. Isto porque para definir o realce de erros no Eclipse é necessário saber a localização dos erros no ficheiro pretendido.

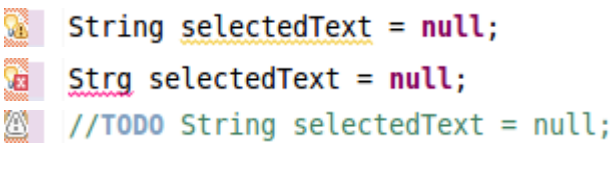
Na representação visual de ficheiros no Eclipse existe o conceito de marcador. Os marcadores têm uma determinada localização e representação visual no ficheiro. Para além, disso os marcadores têm uma natureza persistente, na medida em que ficam associados ao ficheiro até ser ordenada a sua remoção.

O Eclipse disponibiliza vários tipos de marcadores padrão, três exemplos que o leitor poderá reconhecer são os seguintes:

Marcador de aviso (**warning**)

Marcador de erro (**error**)

Marcador de tarefa (**todo**)



Os marcadores possuem representações visuais no texto (ex: sublinhado) e nas barras laterais dos editores no Eclipse, esta última permite ao utilizador uma rápida localização dos marcadores em ficheiros de maior dimensão.

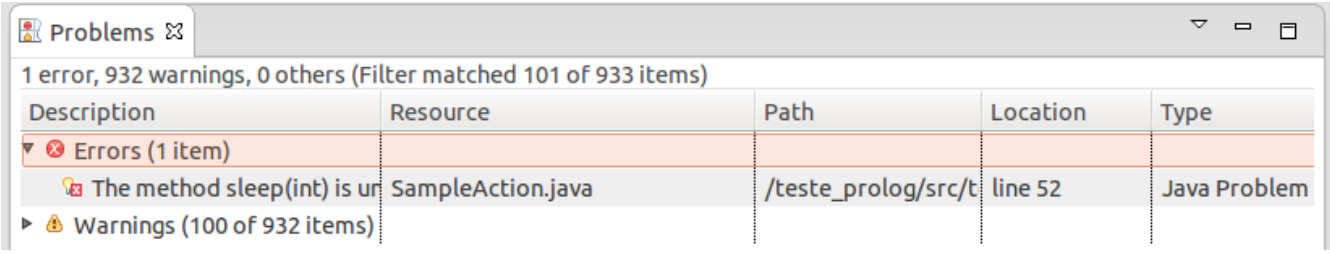


Figura 48 – Problems View do Eclipse

Outra propriedade interessante dos marcadores é o facto dos mesmos serem acessíveis durante a execução do Eclipse, o que permite a sua fácil atualização e até consulta para gerar vistas como a **Problems View** (figura 48) do Eclipse. Esta vista revela informações sobre todas as ocorrências de marcadores de erro e aviso nos projetos do Eclipse.

O CxIDE utiliza os marcadores de erros do Eclipse para o realce de erros.

O único problema é que as informações sobre os erros do ficheiro em edição apenas estavam disponíveis no CxProlog. Era necessário passar essas informações para o Java (secção 6.6). Com esse propósito foi criado um novo predicado **editor_getCodeErrors**.

Neste predicado o Java envia o conteúdo do atual ficheiro em edição no CxEditor, o CxProlog analisa o mesmo e envia o resultado para o Java.

```
editor_getCodeErrors:-
editor_getCurrentContent(Code),code_errors(Code,Errors,java_call('org/eclip
se/cxide/Menu_ops/Editor_Operations','errorsHighlight:([Ljava/lang/Object;)
V',[Errors],_).
```

Eis um diagrama que resume o funcionamento do realce de erros do CxEditor:

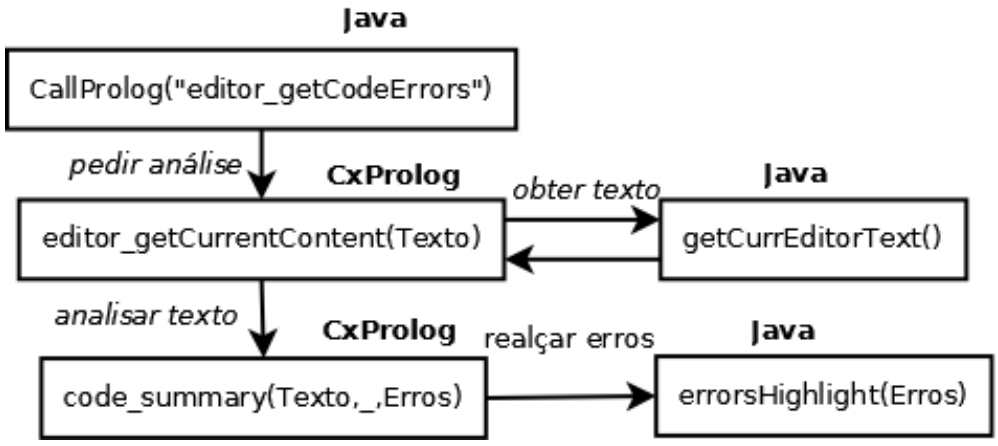


Figura 49 – Funcionamento do realce de erros

O método estático Java:

```
public static void errorsHighlight(Object[] errors)
```

É responsável por processar a lista de erros enviada pelo CxProlog adicionando os respetivos marcadores de erros no atual ficheiro em edição.

Eis um exemplo de como se adiciona um marcador de erros ao atual ficheiro em edição:

```
//Obter o atual ficheiro em edição
IFile file= ResourcesPlugin.getWorkspace().getRoot().getFile(relativePath);
IResource r = file;
//Adicionar um marcado de erros
IMarker problemMarker = r.createMarker(IMarker.PROBLEM);
//Mensagem de erro
problemMarker.setAttribute(IMarker.MESSAGE, msg);
//Localização inicial no ficheiro do erro
```

```
problemMarker.setAttribute(IMarker.CHAR_START, charStart);
//Localização final no ficheiro do erro
problemMarker.setAttribute(IMarker.CHAR_END, offset);
//Caminho relativo para o ficheiro
problemMarker.setAttribute(IMarker.LOCATION, relPath);
```

8.4.6 Colapso e Expansão

Por vezes os ficheiros em edição tornam-se demasiados grandes e complexos. O colapso e expansão dão o poder ao utilizador de organizar o ficheiro à sua vontade de forma a abstrair-se da dimensão ou complexidade de determinadas zonas.

Atualmente o CxEditor dispõe dum documento onde armazena todo o texto em edição. O utilizador visualiza o documento através dum *viewer*.

Para oferecer colapso e expansão no editor é necessário complicar um pouco mais.

A abstração providenciada pelo colapso e expansão é fruto da utilização duma projeção sobre o documento original (figura 50). O utilizador não visualiza diretamente o documento original (sem colapso) mas sim uma projeção do mesmo. Sendo que essa projeção revela as zonas afetadas pelo colapso (30).

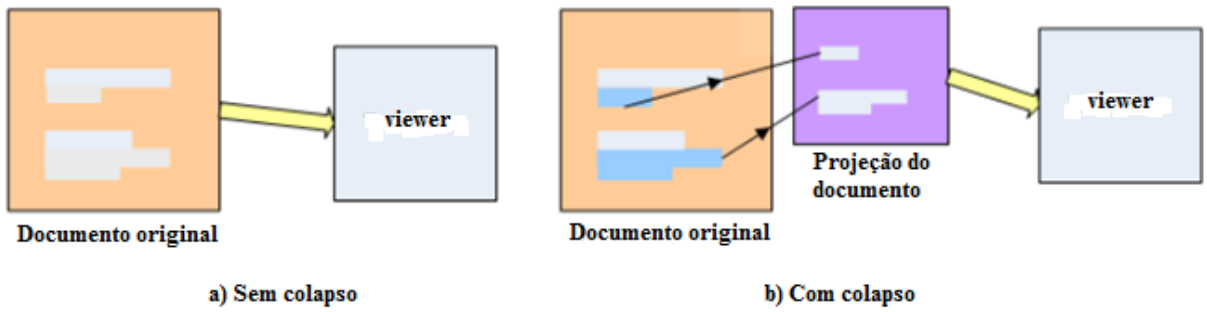


Figura 50 – Diferentes modelos dum Editor sem/com colapso

Este modelo permite manter o documento original ao mesmo tempo que permite uma visualização simplificada do mesmo.

O documento de projeção precisa de informações sobre quais as zonas do documento original são alvo de colapso. Para definir estas informações são utilizadas anotações de projeção representadas por instâncias da classe **ProjectionAnnotation**[37] da plataforma Eclipse.

Uma anotação de projeção pode ser colapsada ou expandida. Se for expandida então corresponde a um segmento do documento original. Se for colapsada, representa uma região do documento de projeção.

Para definir uma anotação de projeção é apenas necessário ter a posição inicial e final da zona do documento a que se pretende possibilitar o colapso.

O **CxEditor** disponibiliza o colapso dos predicados no documento em edição. Para a implementação foi necessário obter a localização inicial e final dos predicados no **CxEditor**.

No realce de erros para obter a posição de erros foi utilizado o predicado **code_summary** do CxProlog.

[37] Mais informações sobre a classe ProjectionAnnotation em: goo.gl/hvltim

O predicado **code_summary/3** retorna duas listas:

Errors - posições de todos os erros (**Errors**) do texto analisado.

Summary - todas as posições dos predicados no texto analisado.

Assim, para obter as posições dos predicados no texto para definir o colapso e expansão foi utilizada a lista **Summary** do **code_summary/3**. O predicado que permite ao Java aceder à essa lista é o seguinte:

```
editor_getCodeFolding:-
editor_getCurrentContent(Code),code_summary(Code,Summary,_),java_call('org/
eclipse/cxide/Menu_ops/Editor_Operations','updateFolding:([Ljava/lang/Objec
t;)V',[Summary],_).
```

Eis o diagrama que resume o funcionamento do colapso:

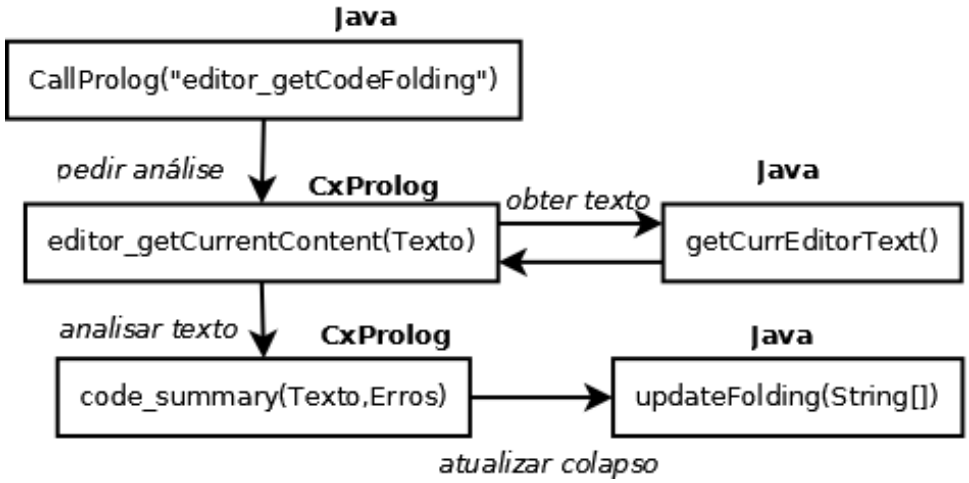


Figura 51 – Funcionamento do colapso

Obtida a lista com as localizações do predicado no Java era finalmente possível definir as anotações de projeção para definir as zonas de colapso. O código que define as anotações é apresentado seguidamente:

```
public static void updateFolding(Object[] predInfo){
      ArrayList<Position> predPositions = new ArrayList<Position>();
      int i=0;
      while(i<predInfo.length){
            int beggining_offset=(int) predInfo[i];
            int line=(int) predInfo[++i];
            int line_offset=(int) predInfo[++i];
            int ending_offset=(int) predInfo[++i];
            int end_line=(int) predInfo[++i];
            int unknown = (int) predInfo[++i];
 predPositions.add(new Position(beggining_offset-1, ending_offset-
                  beggining_offset-1));
      }
      CxEditor.updateFoldingStructure(fPositions);
}
```

```
public static void updateFoldingStructure(ArrayList<Position> positions){
Annotation[] annotations = new Annotation[positions.size()];
//this will hold the new annotations
//with their corresponding positions
HashMap<ProjectionAnnotation,Position> newAnnotations =
    new HashMap<ProjectionAnnotation,Position>();

for(int i =0;i<positions.size();i++){
ProjectionAnnotation annotation = new ProjectionAnnotation();
                      newAnnotations.put(annotation,positions.get(i));
                      annotations[i]=annotation;
}
annotationModel.modifyAnnotations(oldAnnotations,newAnnotations,null;
oldAnnotations=annotations;}
```

8.4.7 Vista estruturada

O **CxEditor** disponibiliza uma vista estruturada (**CxOutline**) que organiza todos os predicados do atual ficheiro em edição. Para além disso, a **CxOutline** possibilita uma navegação rápida para os elementos que exibe. Nesta secção é revelada a implementação da **CxOutline**.

O primeiro passo foi a criação duma nova vista no Eclipse.

Para criar vistas o Eclipse disponibiliza a extensão **org.eclipse.ui.views**[38].

A definição duma nova vista requer uma classe que estenda a classe **ViewPart**[39] disponibilizada pela plataforma Eclipse e com esse propósito foi desenvolvida a seguinte classe:

public class OutlineView extends ViewPart

Uma das formas mais fáceis de definir o conteúdo duma vista no Eclipse é através da definição de uma estrutura em árvore.

Uma árvore é constituída por nós. Os nós, no caso da **CxOutline**, representam termos. Sendo que cada nó têm sub-nós que representam átomos.

Existem duas classes representantes dos possíveis elementos da vista:

```
public abstract class Node
public class PredNode extends Node
public class AtomNode extends Node
```

Ambas as classes que representam nós estendem a classe **Node**. Esta classe representa um nó da árvore. Implementando várias operações como o acesso ao nó pai (*parent node*).

Foi necessário definir que informações representam cada nó. Os nós termo são representados pelo funtor do termo e a sua aridade, enquanto os átomos são apenas representados apenas pelo seu funtor (figura 52).

A parte mais desafiante na implementação da **CxOutline** foi obter as informações sobre os nós. Era necessário processar o atual ficheiro em edição obtendo todos os seus átomos e termos com os respetivos funtores e aridade.

Usando as cláusulas **code_summary/3** e **editor_getCurrentContent/1** foi possível extrair todos os termos do ficheiro em edição. Porém os termos eram extraídos como texto sendo necessária a sua conversão para termos. Felizmente, o CxProlog disponibiliza a cláusula **atom_term/2** que converte

[38] Mais informações sobre a extensão org.eclipse.ui.views: goo.gl/2M0HDp
[39] Mais informações sobre a classe ViewPart: goo.gl/cqbpHm

um termo na sua representação textual ou vice-versa. Sendo assim possível programar uma nova cláusula que converte toda uma lista de elementos textuais numa lista de termos.

```
atomsToTexts([],[]).

atomsToTexts([Atom|Atoms],[Text|Texts]):-
atom_term(Text,Atom),atomsToTexts(Atoms,Texts).
```

Após a obtenção dos termos, faltava obter os seus funtores e aridades. Novamente, o CxProlog disponibiliza uma cláusula ótima para o pretendido. A cláusula **functor(Term,Fun,Ar)** que dado um termo retorna o seu funtor e aridade. Mais uma vez era necessária a troca de várias informações entre Java e CxProlog (secção 6.6). Alguns predicados CxProlog e métodos estáticos Java foram desenvolvidos para lidar com a necessária comunicação.

Seguidamente são sumariados os passos da implementação da CxOutline:

1. Obter texto em edição **editor_getCurrentContent/1**
2. Code summary para obter todos os termos **code_summary/3**
3. Coverter texto em termos **atom_term/2**
4. Extrair funtor e aridade dos termos **functor/3**
5. Construir a árvore.
6. Gerar a vista estruturada.

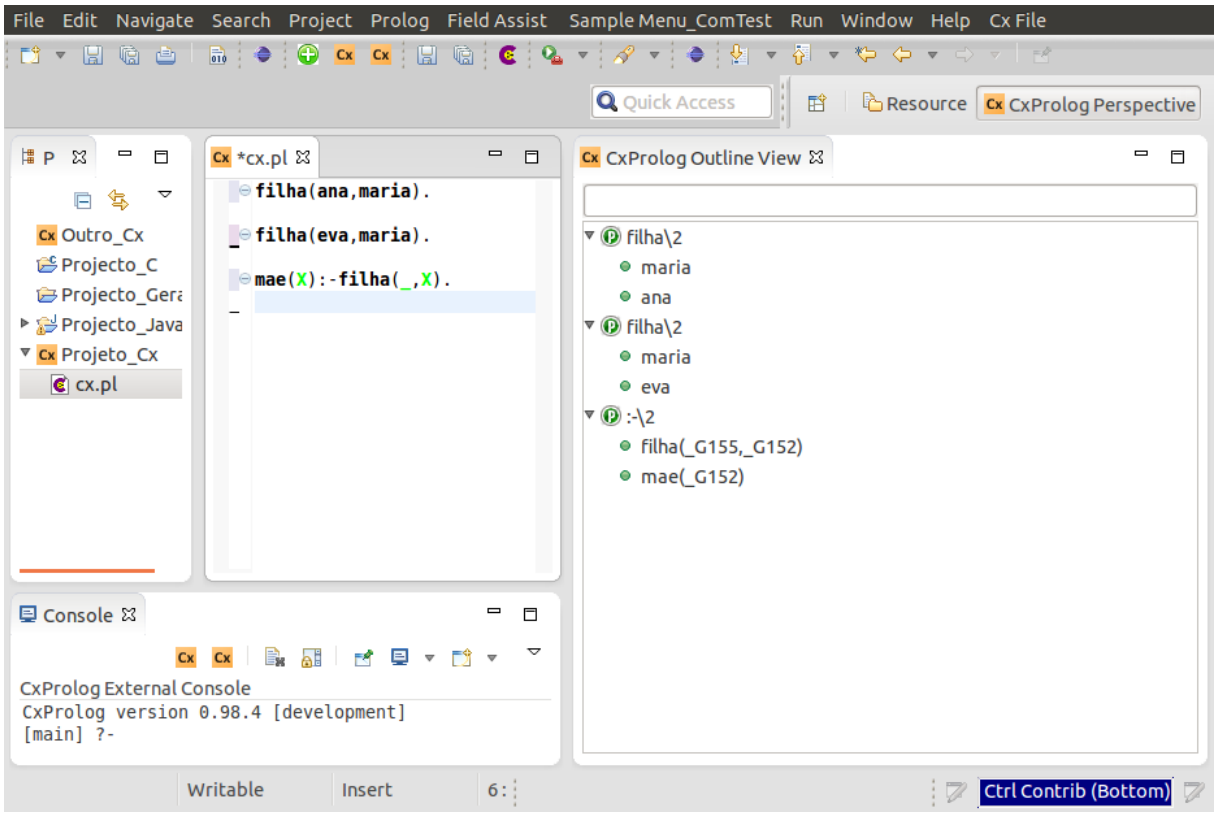


Figura 52 – Exemplo da utilização da CxOutline

8.4.8 Melhorarias à eficiência do CxEditor

Realce de erros, colapso e vista estruturada estavam todos dependentes da cláusula **code_summary**. Isto originou um problema de eficiência. O **code_summary** necessitava receber o texto a ser analisado. Assim, cada vez que era atualizado o realce de erros, colapso e vista estruturada era necessário o envio duma possível "grande *string*" (todo o texto do ficheiro em edição) do Java para o CxProlog. A determinada altura o CxIDE enviava todo o texto em edição três vezes para o CxProlog, isto ocorria quando eram executados os seguintes predicados:

editor_getCodeErrors – Obter informações sobre todos os erros do atual texto em edição no CxEditor. Predicado executado cada vez que o realce de erros era atualizado.

editor_getCodeFolding – Obter a localização de todos predicados do atual texto em edição no CxEditor. Predicado executado aquando da atualização do colapso ou vista estruturada, ou seja era executado duas vezes.

Para melhorar a eficiência do CxIDE, os predicados **editor_getCodeFolding** e **editor_getCodeErrors** foram substituídos pelo predicado **editor_AnalyseCode**.

Atualmente, o CxIDE atualiza o colapso, realce de erro e vista estruturada ao mesmo tempo (secção 8.5.2). Para cada atualização é executada apenas uma vez o predicado **editor_AnalyseCode**, dessa forma é apenas enviada uma vez o texto em edição no **CxEditor** para análise em vez das anteriores três vezes. Esta alteração na implementação permitiu uma melhoria em termos de eficiência.

8.4.9 Momento de atualização de realce de erros, colapso e vista estruturada

O CxIDE, inicialmente, apenas atualizava o realce de erros, colapso e vista estruturada dum ficheiro no CxEditor após ser executado o **SAVE** do mesmo por parte do utilizador.

Em termos de eficiência era uma boa solução, pois todo o processamento envolvido nessas atualizações só era realizado num momento determinado pelo utilizador. Na prática, não era de todo benéfico para o utilizador, a necessidade de dar **SAVE** a um ficheiro para saber se o mesmo tinha erros de sintaxe.

O IDE Java do Eclipse atualiza o seu realce de erros, colapso e vista estruturada cada vez que o utilizador para de digitar algo no editor. O CxIDE ambicionava ter um funcionamento semelhante. Com esse, propósito foi realizado um estudo sobre a solução de implementação do IDE Java.

O IDE para Java do Eclipse utiliza um Reconciler (31) para definir o momento de atualização de funcionalidades relacionadas com a edição. O Reconciler é basicamente uma *thread* que recorda todas as alterações no texto sendo invocada periodicamente (cada vez que o utilizador para de digitar).

Assim foi implementada a classe MyReconciler:

```
public class MyReconciler implements IReconcilingStrategy,
                IreconcilingStrategyExtension
```

Esta classe, permitiu que a atualização de realce de erros, colapso/expansão e vista estruturada fossem realizadas no momento em que o utilizador para de digitar. Este sem dúvida é um dos melhores momentos para atualizar todas as funcionalidades que relacionadas com a edição, não prejudicando de todo o utilizador e não exigindo uma contínua atualização que prejudicaria o desempenho do IDE.

8.5 Consola

Nesta secção é descrita a implementação das consolas do CxIDE.

Criar uma nova consola no Eclipse é bastante fácil, utilizando a extensão **org.eclipse.ui.console**[40].

```
//Criar a nova consola
IOConsole console = new IOConsole("Nova Consola", null);
IConsole consoles[] = new IConsole[1];
consoles[0]=console;
//Adicionar a nova consola ao ambiente
ConsolePlugin.getDefault().getConsoleManager().addConsoles( consoles );
```

[40] Mais informações sobre a extensão org.eclipse.ui.console: goo.gl/AUej2O

Eis a consola criada, pelo código acima:

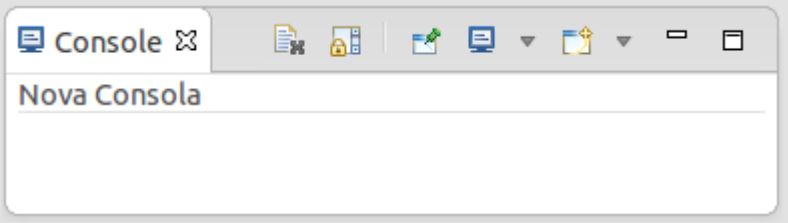


Figura 53 – Exemplo duma nova consola no Eclipse

Uma **IOConsole**, como a da figura 53, disponibiliza dois *stream*:

InputStream – Para aceder às informações digitadas na consola pelo utilizador.

OutputStream – Para enviar informações para a consola.

Para implementar a consola do CxIDE que opera sobre a biblioteca dinâmica foi necessário:

1. Obter o input do utilizador via *inputstream*.
2. Enviar esse input para o CxProlog embebido.
3. Retornar a resposta da execução via *outputstream.*

No entanto, o que ao início parecia simples tornou-se bastante complicado.

A consola do CxProlog requeria a utilização de duas *threads*. Uma para ler o *input* do utilizador e enviar o mesmo para o CxProlog (biblioteca dinâmica). Outra para ler o *output* do CxProlog e enviar o mesmo para a consola (figura 54).

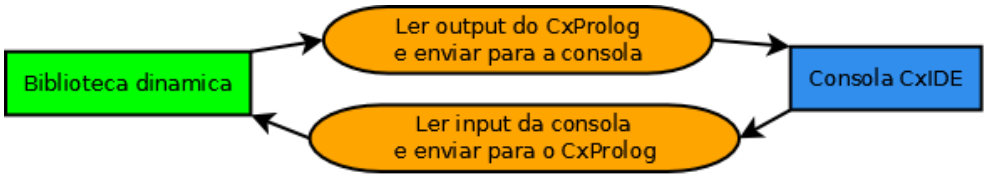


Figura 54 – Diagrama das *threads* Java responsáveis com a interação com a consola interna do CxIDE

É importante nesta altura esclarecer qual é arquitetura de threads do Eclipse. Por omissão, o Eclipse usa uma única thread para executar todas as instruções de código. Esta thread executa o ciclo de eventos da aplicação e é a única thread que é permitida interagir com a interface do utilizador (UI). É chamada thread principal.

Todos os eventos da interface do utilizador são executados um após o outro. Se for realizada uma operação de longa duração na thread principal, a aplicação não responderá à interação do utilizador durante o tempo de execução da operação.

O CxIDE executa na thread principal do Eclipse, logo a não utilização das *threads* da figura X bloquearia o CxIDE aquando do bloqueamento duma leitura. No entanto, a própria utilização de mais threads traria novos problemas.

Inicialmente, a única forma de o CxIDE contactar com a biblioteca dinâmica do CxProlog era através do **Prolog.CallProlog**, portanto para executar o *input* do utilizador era necessário usar o **Prolog.CallProlog(input)**. O CxIDE e respectiva biblioteca dinâmica executam na *thread* principal não existindo problema em realizar o **CallProlog** no CxIDE. No entanto, agora era necessário executar **CallProlog** numa *thread* diferente, especificamente, na *thread* responsável por ler input do utilizador e enviar o mesmo para o CxProlog. Surgia aqui um problema de reentrância.

Um código reentrante é código que pode ser acedido por um outro "ator" antes de uma prévia inovação ter terminado, sem ser afetado o caminho que o primeiro "ator" teria seguido no código. Ou seja, é possível reentrar no código em execução continuando a produzir resultados correctos.

A biblioteca dinâmica do CxProlog era acedida por duas diferentes *threads*, gerando um problema de reêntracia. A biblioteca detetava a reentrância originando um erro e terminando a sua execução, impossibilitando o funcionamento do CxIDE.

Existia a possibilidade de implementar a consola do CxIDE com leituras não bloqueantes, sem a necessidade de *threads*. Mas neste cenário continuavam a existir problemas:

- CxIDE ficaria bloqueado até à conclusão da execução dum golo.
- A consola não ofereceria uma interação semelhante à execução do CxProlog num terminal. Isto porque a consola estava assente no **CallProlog** e neste sentido não existia a noção de *prompt* nem a possibilidade de *backtracking*.

Os problemas apontados atrasaram bastante a implementação da consola e com o âmbito de avançar foi decidido criar uma consola com uma implementação semelhante à consola do PDT (secção 3.2.1). O resultado dessa implementação seria a consola externa do CxIDE.

8.5.1 Consola Externa

O PDT não utiliza bibliotecas dinâmicas mas sim instalações locais dos Prologs que suporta. Por exemplo, o PDT lança um processo **SWI-PROLOG** e utiliza o mesmo como escravo da sua consola.

Seguindo o exemplo da implementação da consola do PDT, foi lançado um processo CxProlog (duma instalação local) e utilizado o mesmo como "escravo" da consola do CxIDE.

Esta solução originou a denominada **CxProlog External Console**, a consola externa do CxIDE que pode executar sobre uma instalação local do CxProlog. Foi igualmente desenvolvida uma nova página de preferências relacionada com o CxIDE que permite a seleção da instalação local do CxProlog a ser utilizada.

Na classe **CxExternalConsole** é realizada toda a implementação da consola externa. Nesta classe o processo CxProlog da instalação local é lançado através duma instância da classe ProcessBuilder disponibilizada pelo Java. O ProcessBuilder permite criar processos do sistema operativo. Após a criação do processo o CxIDE usa um Socket (java.net.Socket) para comunicar com o processo. Sendo os streams (input, output) do Socket conectados ao streams da consola externa.

No entanto, existiu a ambição de levar a consola externa do CxIDE mais longe do que a consola do PDT, possibilitando ao utilizador a realização de uma fácil conexão a um servidor de CxProlog. A implementação da conexão a um servidor foi bastante semelhante à do processo local, residindo a diferença no não lançamento do processo, sendo novamente usando sockets para conectar ao servidor. O utilizador apenas precisa de preencher um simples diálogo (secção 7.2) para conectar a consola externa ao servidor.

A consola externa enriqueceu o projeto ao possibilitar ao utilizador do CxIDE uma rápida mudança do CxProlog utilizado na consola sem a necessidade de realizar qualquer atualização ao *plugin*.

Porém não era de todo pretendido que a única consola disponibilizada no CxIDE fosse a consola externa. A mesma executava sobre uma instalação local ou externa do CxProlog e não sobre a biblioteca dinâmica o que acarretava as seguintes consequência:

- Impossibilidade de executar extensões ou configurações ao CxIDE durante a execução do Eclipse.
- Inexistência de acesso à biblioteca dinâmica do CxProlog, durante a execução do Eclipse.

Obviamente várias ambições do projeto estavam em causa caso a única consola do CxIDE fosse a consola externa.

8.5.2 Consola Interna

Era vital para as ambições do projeto a existência duma consola que executasse sobre a biblioteca dinâmica do CxProlog. Apareceram vários problemas inicialmente mas agora surgia uma possível solução.

O criador do CxProlog, Prof.Artur Miguel Dias encarou os problemas que surgiram na implementação da consola, descritos no inicio da secção 8.5, como uma oportunidade: A oportunidade para enriquecer as funcionalidades do CxProlog relacionadas com corrotinas. Não estando disponíveis *threads* no CxProlog resolveu-se implementar a consola interna usando corrotinas.

Vários predicados foram melhorados ou adicionados para trabalhar com as corrotinas em CxProlog, nomeadamente:

```
%Cria uma nova corrotina CxProlog que executa o dado predicado

thread_new(Nome_Routina, Predicado, fail)

%Dá algum tempo de execução a uma determinada corrotina. Se retornar true
significa que a corrotina necessita de mais tempo de execução, caso
contrário a corrotina não necessita de tempo de execução porque está
bloqueado à espera de input.

coroutiningRunABit(Nome_Rotina)

%Envia input para uma routina sobre a forma duma string

coroutiningInputLine(Nome_Rotina, Input)

%Lê o output da corrotina

coroutiningOutputText(Nome_Rotina)
```

O leitor atento terá reparado que uma corrotina executa um determinado predicado (ver *thread_new* acima). No caso da corrotina da consola interna o predicado que é executado chama-se top_level/0 e implementa a interação normal com o utilizador. Isto é, imprime inicialmente um *prompt* e depois entra num ciclo onde aguarda input para executar, após a execução do input imprime o resultado para os canais definidos seguido do *prompt*.

Criadas as condições era altura de implementar a consola do CxIDE que operava sobre a biblioteca dinâmica. Para isso foi criada a seguinte classe:

```
public class CxInternalConsole extends IOConsole
```

Esta classe adiciona uma nova consola ao CxIDE com o nome de "CxProlog Internal Console".

Na classe **CxInternalConsole** é criada uma corrotina chamada **cxRoutine** e duas *threads* Java:

1º Thread - *ReadUserWriteCxProlog*

Thread Java responsável por ler o *input* na consola interna e enviar o mesmo para a **cxRoutine**.

Esta *thread* realiza uma espera ativa pelo *input* do utilizador na consola interna, a figura 55 revela o funcionamento da thread.

O *input* é guardado até ser detetada uma quebra de linha (enter do utilizador), nesse momento o *input* guardado é enviado para a **cxRoutine** usando a coroutiningInputLine. Após o envio do *input*, é dado tanto tempo de execução quanto necessário à **cxRoutine** usando **coroutiningRunABit** para que possa ser processado o *input* enviado. Quando a **cxRoutine** termina o processamento do *input* é executada a *thread* **ReadCxPrologWriteCxProlog**.

2º Thread - *ReadCxPrologWriteConsole*

Thread Java responsável por ler o *output* da **cxRoutine** e enviar o mesmo para a consola interna, a figura 56 revela o funcionamento da thread.

Esta *thread* é executada sempre que a **cxRoutine** termine o processamento de *input*. A função principal desta *thread* é utilizar **coroutiningOutputLine** para ler todo o *output* da **cxRoutine**, sendo esse *output* enviado para a consola interna usando os *streams* adequados.

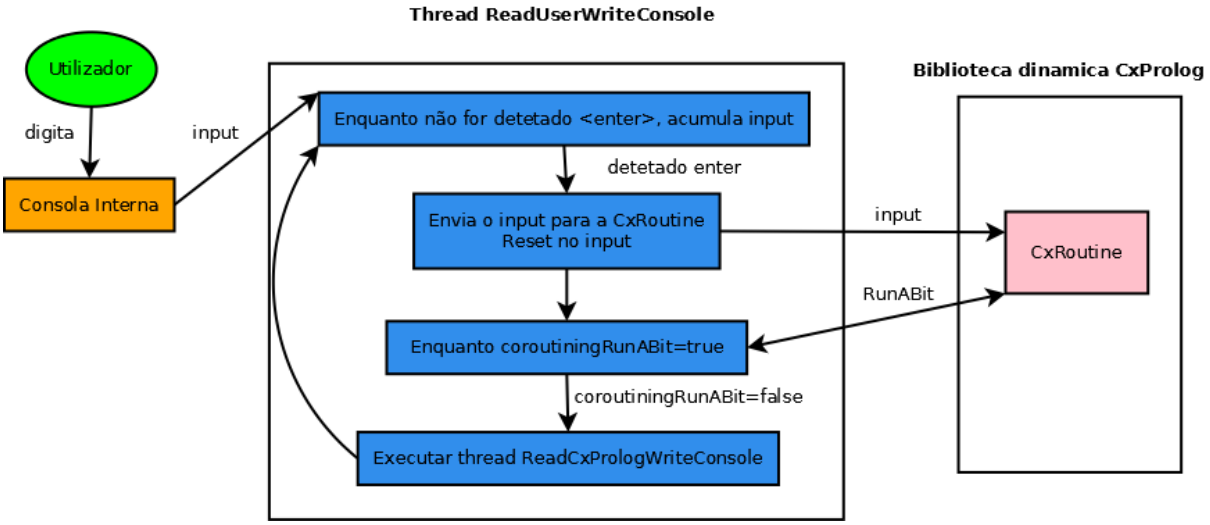


Figura 55 – Diagrama de funcionamento da Thread ReadUserWriteConsole

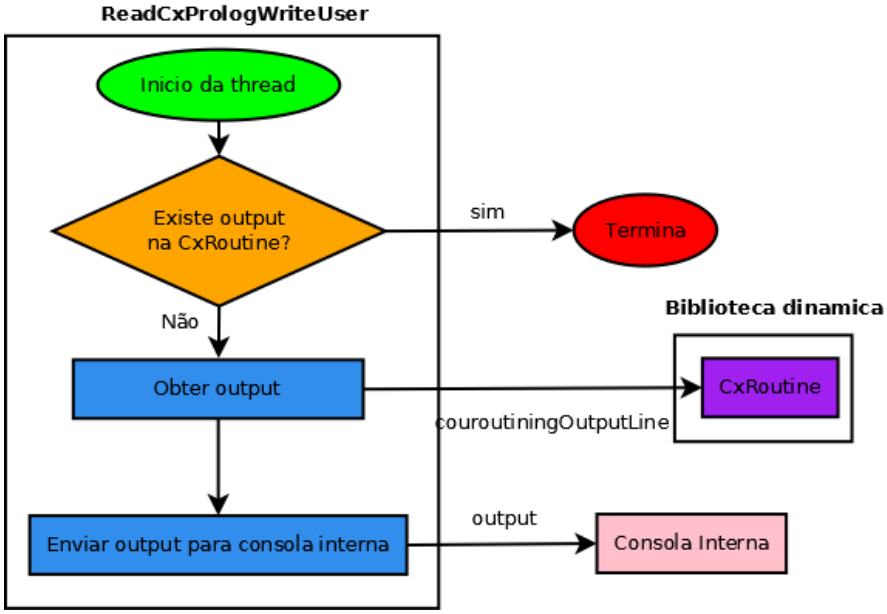


Figura 56 – Diagrama de funcionamento da *thread* ReadCxPrologWriteUser

A cxRoutine executa o predicado top_level/0 que foi descrito anteriormente oferecendo uma comunicação semelhante à execução do CxProlog num terminal (figura 57).

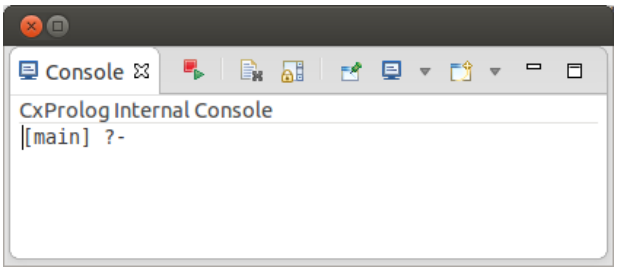


Figura 57 – Consola Interna do CxIDE

Assim uma importante parte do projeto estava concluída. O utilizador do CxIDE disponha duma consola que executava sobre a biblioteca dinâmica do CxProlog, que possibilitava:

- ✓ Execução dinâmica de extensões e/ou configurações ao CxIDE/Eclipse.
- ✓ Feedback automático de menus e diálogos declarados em CxProlog.
- ✓ Que a consola e o CxIDE operassem sobre o mesmo estado da biblioteca dinâmica.

Com a implementação a consola dinâmica do CxIDE tornava-se mais fácil testar o funcionamento do CxIDE o que acelerou o desenvolvimento do projeto.

8.6 Perspetiva

O CxIDE já dispõe do CxEditor de duas consolas e da CxOutline. A perspetiva CxPerspective foi criada para organizar todos componentes do CxIDE. Ao selecionar a CxPerspective, o utilizador terá seu ambiente de trabalho Eclipse totalmente preparado para a utilização do CxIDE. Isto é o CxEditor numa posição central, à sua esquerda o navegador de projetos, na parte inferior a consola, e à direita a vista estruturada (Tabela J).

<table>
<tr><td colspan="3">Barra de menus e ferramentas do Eclipse</td></tr>
<tr><td>Navegador de projetos</td><td>CxEditor</td><td>CxOutline</td></tr>
<tr><td colspan="3">CxConsole</td></tr>
</table>

Tabela 8 – Esquema da CxPerspective

Para criar perspetivas o Eclipse disponibiliza a extensão **org.eclipse.ui.perspectives** (figura 58).

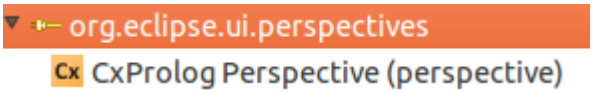


Figura 58 – Extensão utilizada para a criação da CxPerspective

Para definir a organização duma determinada perspetiva o Eclipse disponibiliza a extensão: **org.eclipse.ui.perspectiveExtensions** (figura 59)**.**

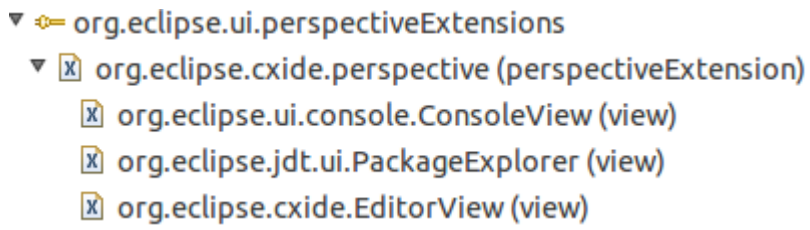


Figura 59 – Extensão utilizada para definição da organização da CxPerspective

Nem é preciso programar para definir uma nova perspetiva, basta utilizar a framework do PDE para declarar quais as vistas que compõe a perspetiva e a localização das mesmas.

8.6.1 Natureza de projeto

Existir uma noção de projeto CxProlog era importante não só para uma identificação inequívoca como pela possibilidade de adicionar opções especificamente só aos projetos CxProlog.

O Eclipse já dispõe de várias distintas naturezas de projetos (figura 60).

Project Nature's ID	Description
org.eclipse.jdt.core.javanature	Java Projects
org.eclipse.buildship.core.gradleprojectnature	Gradle Projects
org.eclipse.m2e.core.maven2Nature	Maven Projects
org.eclipse.pde.core.org.eclipse.pde.PluginNature	Eclipse Plugin Projects
org.eclipse.pde.core.org.eclipse.pde.FeatureNature	Eclipse Feature Projects
org.eclipse.pde.core.org.eclipse.pde.UpdateSiteNature	Eclipse Updatesite Projects

Figura 60 - Exemplos de naturezas de projeto do Eclipse

A natureza dum projeto CxProlog é denominada “CxNature”.

Para criar naturezas de projeto o Eclipse disponibiliza a extensão:

org.eclipse.core.resources.natures

É necessária a definição duma classe que implemente a **IProjectNature** e nesse sentido foi criada a:

```
public class CxProject implements IProjectNature
```

Não existe nada a realçar sobre a implementação da classe CxProject, pois não era pretendida nenhuma configuração especial para os CxProject (projetos com a natureza CxNature).

Para uma fácil identificação visual no navegador de projetos dos CxProjects foi utilizada a extensão:
org.eclipse.ui.ide.projectNatureImages

Para associar um ícone específico aos CxProjects (figura 61).

Figura 61 - CxNature entre outras diferentes naturezas de projetos

A CxNature estava concluída e assim era possível distinguir um projeto de CxProlog de todos os outros.

9 Avaliação do Sistema

Importante realçar que não existe nenhum método ou critérios "oficiais" para avaliar a qualidade de um IDE. Indubitavelmente, o sucesso de um IDE está diretamente relacionado com a opinião dos seus utilizadores. No final de contas, um IDE têm sucesso se a comunidade para o qual foi desenvolvido apreciar o mesmo. Assim, para obter uma noção da qualidade do CxIDE foram realizados testes com utilizadores. Algo também importante a avaliar era o desempenho da solução de implementação que envolveu corrotinagem. Nesse sentindo foi realizado um *benchmark* no CxIDE para avaliar o seu desempenho.

9.1 Testes com utilizadores

Os testes com utilizadores do CxIDE foram realizados na SQIMI, em que três colaboradores da empresa dedicaram, no mínimo, cerca de 45 minutos à instalação e experimentação do CxIDE.

Os utilizadores receberam o manual de instalação e um guia rápido de utilização. Em especial, o guia sugeria alguns exemplos práticos que o utilizador poderia seguir de forma a explorar várias funcionalidades do IDE.

Foi dada completa liberdade aos utilizadores para experimentar o IDE durante uma semana. Os utilizadores poderiam usar o guia se sentissem necessidade para tal.

Após uma semana de acesso ao CxIDE, foi entregue aos utilizadores um questionário. Seguidamente serão reveladas todas as questões colocadas, o motivo das mesmas e os seus resultados:

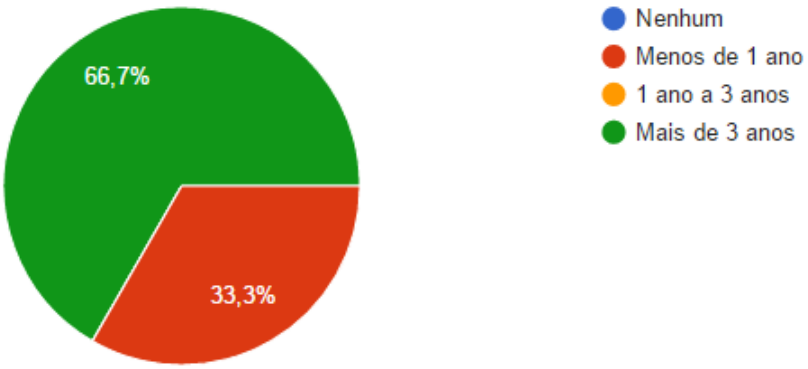
Q1: Tempo de experiência com o CxProlog?

Possíveis respostas:

- Nenhum
- Menos de um 1 ano
- 1 ano a 3 anos
- Mais de 3 anos

Motivo: Era importante perceber se a experiência sobre o CxProlog seria um fator diferenciador na sua apreciação por parte dos utilizadores. A apontar que a SQIMI desenvolve muito do seu *software* recorrendo ao CxProlog. Assim era de esperar que todos conhecessem minimamente o CxProlog.

Resultados:



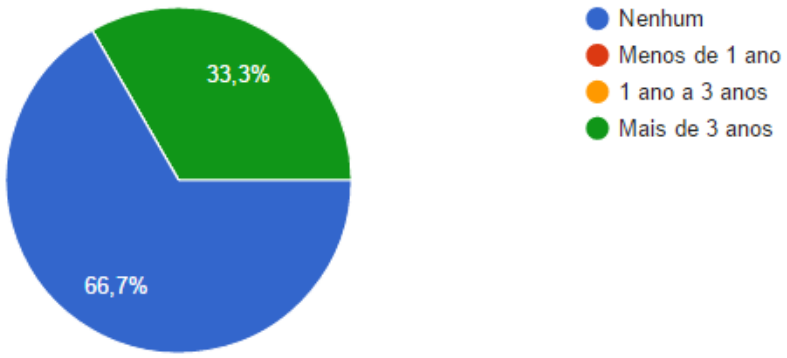
Q2: Tempo de experiência com o Eclipse?

Possíveis respostas:

- Nenhum
- Menos de um 1 ano
- 1 ano a 3 anos
- Mais de 3 anos

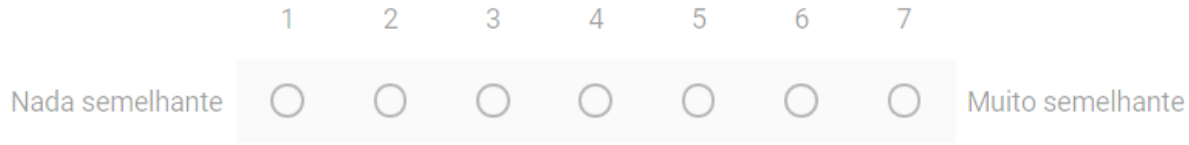
Motivo: Utilizadores menos familiarizados com o Eclipse poderiam ser afetados pela sua curva de aprendizagem o que afetaria a sua opinião sobre o CxIDE, enquanto utilizadores bastante familiarizados com Eclipse poderiam dar a sua opinião sobre o sucesso da imitação, isto é quão semelhante é o CxIDE a outros IDEs para Eclipse que o utilizador já tenha utilizado.

Resultados:



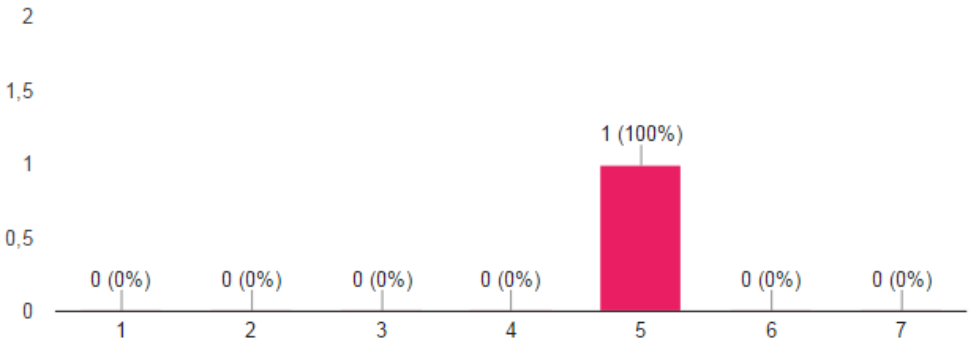
Q3: Se não respondeu "Nenhum" à pergunta anterior. O quão semelhante é a interface do CxIDE com os outros IDEs Eclipse em que já trabalhou?

Possíveis respostas: Uma escala linear de 1 a 7. Em que 1 significa nada semelhante e 7 significa muito semelhante.



Motivos: Avaliar o sucesso da imitação do CxIDE, ou seja o quão semelhante o CxIDE é a outros IDEs para Eclipse.

Resultados:



Q4: Quais as ferramentas de desenvolvimento que atualmente utiliza na programação de CxProlog?

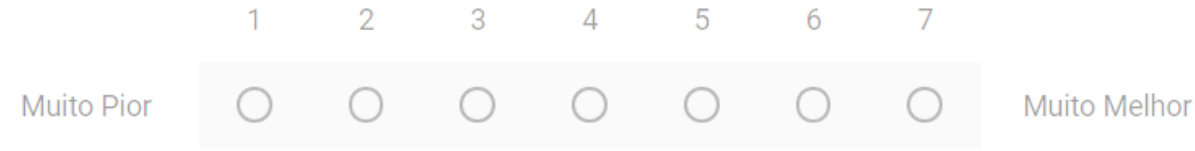
Possíveis respostas: Texto livre

Motivos: Era importante perceber o que atualmente os utilizadores usam para programar em CxProlog. Dessa forma, é possível comparar os seus atuais ambientes de desenvolvimento com o CxIDE.

Resultados: Os utilizadores responderam *emacs* ou *joe* para edição e o terminal para execução.

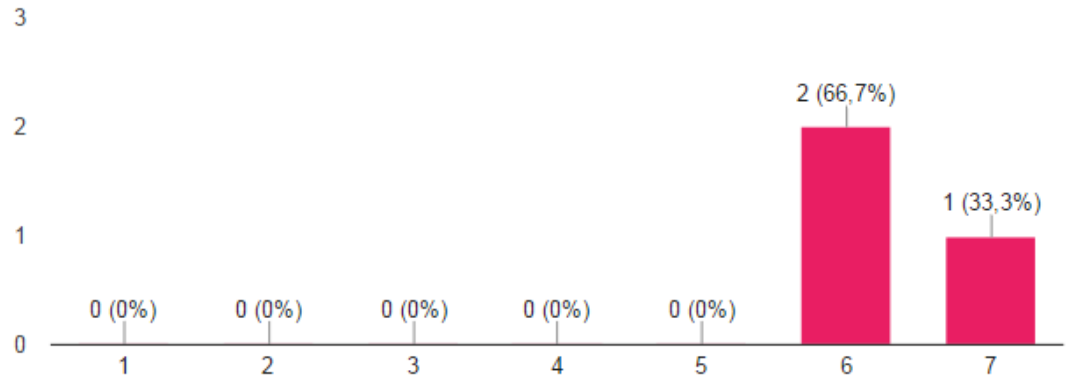
Q5: Comparado ao seu atual ambiente de programação de CxProlog, como classificaria o CxIDE?

Possíveis respostas: Escala linear de 1 a 7. Em que 1 significa muito pior e 7 significa muito melhor.



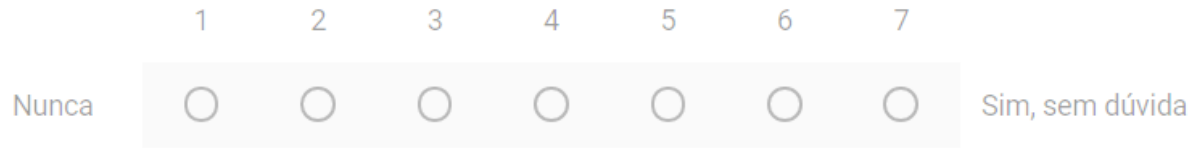
Motivo: Uma das questões mais importantes para a avaliação do CxIDE. As respostas revelariam um importante o grau de apreciação do CxIDE, por parte da sua comunidade alvo, quando comparado aos seus competidores.

Resultados:



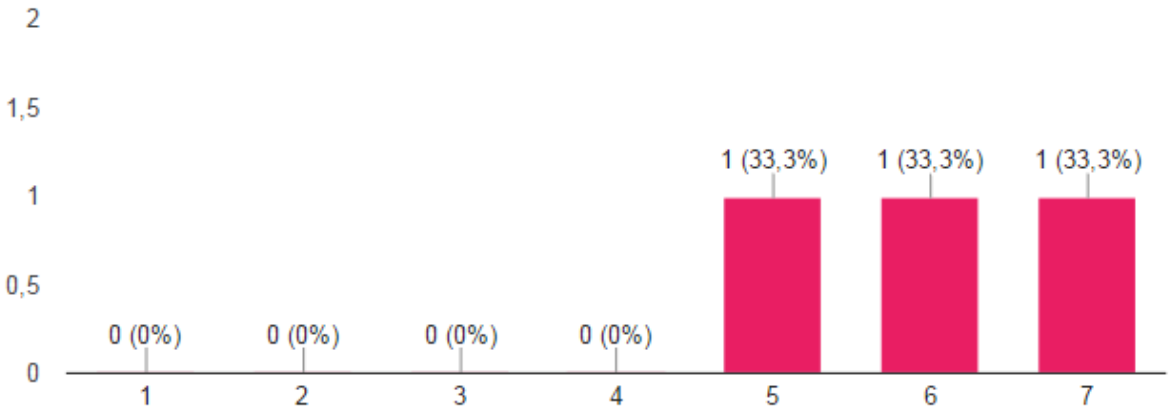
Q5: Consideraria a utilização do CxIDE como o seu principal ambiente de programação para CxProlog?

Possíveis respostas: Escala linear de 1 a 7. Em que 1 significa nunca e 7 significa sim, sem dúvida.



Motivos: A pergunta mais importante para a valorização do CxIDE, já que os utilizadores podem considerar o CxIDE mais vantajoso que as suas atuais ferramentas para a programação mas decidir permanecer nas mesmas (muitas vezes por hábito).

Resultados:



9.1.1 Inquérito pessoal aos utilizadores

Durante o período de testes, todos os utilizadores que testaram os CxProlog foram inqueridos, pessoalmente, sobre quais as suas sugestões de melhorias e problemas detetados. Este inquérito, foi dos momentos mais frutíferos de todo o projeto, permitiu encontrar alguns problemas que não tinham sido detetados (e dificilmente o seriam) aquando do desenvolvimento, tal como surgiram diversas sugestões de como melhorar o CxIDE, tanto em termos de usabilidade (ex: usar ícones mais representativos) como de funcionalidade (ex: implementar um find).

9.2 Conclusões dos testes com utilizadores

Os testes apenas envolveram utilizadores com experiência em CxProlog. No entanto, essa experiência variava, alguns utilizadores tinham pouca experiência (menos de 1 ano) enquanto outros tinham bastante experiência (mais de 3 anos).

Em termos de experiência do Eclipse, apenas um dos utilizadores revelou ser familiarizado com o Eclipse. A resposta positiva (5 numa escala de 1 a 7 em que 1=nada semelhante e 7=muito semelhante) deste utilizador à questão "O quão semelhante é a interface do CxIDE com os outros IDEs Eclipse em que já trabalhou?" revelou que o CxIDE conseguiu alcançar com relativo sucesso ou seu objetivo de imitação.

Todos os utilizadores consideraram o CxIDE bastante melhor que as suas atuais ferramentas de programação ao responderem 6 e 7 numa escala de 1 a 7 em que 1=muito pior e 7=muito melhor. Com estas respostas o CxIDE ganham um grande destaque comparado aos seus possíveis "competidores" (emacs, joe, terminal).

Todos os utilizadores responderam positivamente à pergunta: "Consideraria a utilização do CxIDE como o seu principal ambiente de programação para CxProlog?".

Tendo em conta os testes individualmente é possível ver que os utilizadores com menos experiência em CxProlog estão mais recetivos à utilização do CxIDE como seu principal ambiente de programação. O anterior facto poderá estar relacionado com os hábitos, muitas vezes utilizadores habituados durante vários anos a determinadas ferramentas revelam algum atrito a mudar para outras, mesmo que considere a nova opção melhor. Também não deve ser esquecido o facto de que a curva de apredizagem do IDE pode levar os utilizadores a estarem mais reticentes quanto a uma possível utilização primária do CxIDE para a programação.

Concluindo, os testes com utilizadores foram extremamente positivos, o CxIDE revelou ser um sucesso ao cativar bastante a atenção e interesse da sua comunidade alvo. Adivinha-se um bom futuro para o CxIDE em que o mesmo poderá tornar a programação mais produtiva e confortável para todos os programadores de CxProlog.

9.3 Avaliação da técnica de corrotinagem utilizada

O CxProlog não suporta multithreading. Suporta apenas o mecanismo, mais pobre, das corrotinas. No CxIDE houve o desejo de introduzir uma consola e isso idealmente implicaria o uso de duas threads: uma dedicada a correr o top level do CxProlog na consola e outra dedicada a gerir os eventos da GUI. Não havendo threads, tentou-se resolver da melhor maneira possível usando corrotinas. Isso implicou uma solução bastante elaborada, constituida pelos seguintes elementos:

- Foi criada uma corrotina específica para correr o toplevel. Portanto ficam duas atividades a correr, o CxProlog base mais a corrotina.
- No ciclo de tratamento do SWT foi criada uma atividade "idle" que faz correr a corrotina do toplevel um décimo de segundo de cada vez.
- Simultaneamente, as ações do utilizador sobre a interface gráfica geram chamadas ao CxProlog de base.

Esta solução levanta algumas dúvidas:

- ? Será que o tempo atribuído de um décimo de segundo é excessivo e compromete a agilidade do sistema a responder aos eventos do utilizador?
- ? Será que o tempo atribuído é demasiado reduzido para o Prolog poder produzir respostas atempadas?

Para responder as estas dúvidas preparou-se a seguinte simples experiência, tomamos um programa prolog que demora cerca de 10 segundos a correr e executámo-lo em três situações diferentes:

- Execução num terminal fora do IDE, para determinar qual a velocidade máxima de execução permitida pela máquina.

  Resultado: 13666762.59 lips for 38300 iterations taking 1.39 secs

- Execução na consola do IDE tendo o cuidado de não gerar quaisquer eventos no ambiente gráfico.

  Resultado: 12539274.54 lips for 1373000 iterations taking 54.31 secs

- Execução na consola do IDE mas tentado perturbar ao máximo a atividade da corrotina gerando manualmente uma sequência rápida e ininterrupta de eventos através do rato e do teclado.

  Resultado: 10131032.43 lips for 1373000 iterations taking 67.22 secs

Nos resultados revelados, LIPS significa "Logical Inferences Per Second" e corresponde à medida habitual de velocidade de execução das implementações de Prolog. Quando maior for o valor, mais rápida será a implementação. Varia na razão inversa do tempo de execução.

Cálculo final de resultados:

12539274.54 /13666762.59=.9175 10131032.43/13666762.59 =.7413

A conclusão é que correr um programa Prolog na consola, mantêm um nível de velocidade entre 74% a 92% da velocidade básica da máquina. O que parece ser bastante bom e concluímos que a solução é razoável. Na prática o sistema pode ser utilizado sem ser percetível qualquer tipo de atraso.

Se por ventura os resultados tivessem sido menos favoráveis, a única coisa que se podia fazer para melhor a questões discutidas seria testar outros valores para o Prolog testado. E no limite poderia se ter de prescindir da consola. Pode se ainda fazer a seguinte pergunta porque razão é que usa 2 duas corrotinas e não dois processos diferentes. A resposta é que só com corrotinas ou *threads* se pode oferecer ao utilizador um sistema bem integrado em que determinados comandos pode ser dados através dos menus ou através da consola. Ou seja para haver máxima integração é importante que ambas as corrotinas partilhem o acesso às funcionalidades do IDE.

10 Conclusão

O CxIDE, um IDE baseado em Eclipse para CxProlog, foi o resultado final desta dissertação. Não se pouparam esforços na implementação das funcionalidades do CxIDE, tornando o mesmo num IDE moderno. Moderno, no sentido em que disponibiliza todas as funcionalidades que qualquer utilizador espera encontrar, atualmente, num IDE. Para além disso, é muito provavelmente o único IDE para Eclipse que permite modificar ambiente do Eclipse utilizando CxProlog. O utilizador do CxIDE pode criar novos menus, diálogos e configurar o Eclipse (ex: definir o seu próprio realce de sintaxe) usando apenas o CxProlog.

Para o sucesso deste projeto, revelaram-se essenciais os conhecimentos adquiridos durante todo o "Mestrado de Engenharia Informática", com especial foco para as cadeiras de:

- Programação Lógica – Conhecimento na programação de Prolog.
- Programação Orientada a Objetos – Conhecimentos na programação de Java.
- Metodologias de Desenvolvimento de Software – Conhecimentos na organização dum grande projeto.
- Conceitos e Tecnologias XML – Conhecimentos sobre XML.

Pessoalmente, esta dissertação foi muito benéfica. Foi o maior e mais desafiante projeto de engenharia que desenvolvi dura a minha vida académica. Este projeto permitiu-me adquirir conhecimentos sobre a plataforma Eclipse, interoperabilidade e organização pessoal que enriquecerão bastante o meu currículo.

Finalizando, este projeto de dissertação foi um sucesso, todos os objetivos iniciais foram alcançados, o produto final é muito utilizável e seguramente será bastante útil para todos os programadores de CxProlog.

10.1 Obter o CxIDE

A instalação, projeto e toda a sua documentação (manual de instalação e de utilização) do CxIDE podem ser descarregados ou acedidos nos seguintes links:

GitHub: https://github.com/andreframos/CxIDE

Dropbox;
https://www.dropbox.com/sh/8zh0t9j3nh62w00/AAAL7BVXOE2YQ1ngw64T1Diga?dl=0

É no entanto aconselhada a utilização do GitHub pois não só é possível selecionar os ficheiros que se pretendem descarregar (na dropbox é tudo descarregado num único zip) como no GitHub o utilizador encontrará sempre a versão mais atualizada do CxIDE (o GitHub recebe as atualizações primeiro).

Como está na documentação, problemas ou sugestões relacionadas com o CxIDE podem ser enviadas para o seguinte e-mail: af.ramos@campus.fct.unl.pt

Importante realçar, que é o CxIDE é código aberto, os interessados poderão obter o projeto nos links acima, e alterações benéficas para o CxIDE realizadas por outros poderão vir a ser integradas no projeto original.

10.2 Trabalho futuro

O CxIDE pode ser melhorado e futuramente alguns objetivos serão:

- **Expansão das funcionalidades oferecidas**

  Desenvolvimento de novas funcionalidades, por exemplo um grafo de chamadas.

- **Internacionalização**

  Preparar o CxIDE para ser traduzido para diferentes linguagens.

- **Adição de novos predicados que permitam alterar o Eclipse**

  Atualmente só existem predicados que permitem a criação de diálogos de forma operacional. Poderão ser criados predicados que permitam a criação de diálogos de forma declarativa. E predicados que permitam utilizar primitivas gráficas (ex: draw_line(X,Y)).

- **Suporte**

É previsível que durante os primeiros meses de utilização do CxIDE, por parte da comunidade de programados CxProlog, surgirá importante feedback sobre *bugs*, melhorias, etc. Tentarei dar resposta a esse feedback modificado e lançando novas versões do CxIDE de acordo, até ao final da primavera de 2017.

11 Bibliografia

1. **REBELLABS.** Developer Productivity Report 2013 – How Engineering Tools & Practices Impact Software Quality & Delivery. [Online] ZEROTURNAROUND, 18 de 11 de 2013. [Citação: 27 de 12 de 2015.] http://goo.gl/4w9EHV.

2. **Carbonnelle, Pierre.** TOPIDE Top IDE index. [Online] 2016. [Citação: 09 de 02 de 2016.] https://pypl.github.io/IDE.html.

3. **Booch, Grady.** Collaborative Development Environments. s.l. : IBM Rational, 2006.

4. **Wang, Yi, et al., et al.** Characterizing Developer Behavior in Cloud Based IDEs.

5. Formal-IDE 2016. [Online] Formal Integrated Development Environment (F-IDE), 2016. [Citação: 27 de 11 de 2016.] https://sites.google.com/site/fideworkshop2016/.

6. **Ponzanelli, Luca, Bachelli, Alberto e Lanza, Michele.** Seahawk: Stack Overflow in the IDE. 2013.

7. *Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions.* **Rahman, Mohammad, Yeasmin, Shamina e Roy, Chanchal.** 2014 : s.n.

8. **Foundation, Eclipse.** Eclipse MarkerPlace - Programming Languages. [Online] Eclipse, 2016. [Citação: 27 de 11 de 2016.] https://marketplace.eclipse.org/category/categories/programming-languages.

9. **Microsoft.** Visual Studio Code - Programming languages. [Online] Microsoft, 11 de 2 de 2016. [Citação: 27 de 11 de 2016.] https://code.visualstudio.com/Docs/languages/overview.

10. *Eclipse: A platform for integrating development tools.* **Rivières, J e Wiegand, J.** 2, 2004, IBM SYSTEMS JOURNAL, Vol. 43.

11. **Eclipse.** Eclipse Top Favorites. [Online] [Citação: 19 de 09 de 2016.] https://marketplace.eclipse.org/favorites/top.

12. **Moir, Kim.** The Architecture of Open Source Applications. [Online] [Citação: 08 de 02 de 2016.] http://www.aosabook.org/en/eclipse.html.

13. **Microsoft.** Microsoft Visual Studio. [Online] Microsoft, 2016. [Citação: 27 de 11 de 2016.] https://www.visualstudio.com.

14. **Cogswell, Jeff.** Visual Studio vs Eclipse: A Programmer's Matchup. [Online] [Citação: 09 de 02 de 2016.] http://goo.gl/y3L9dJ.

15. **Oracle.** NetBeans. [Online] Oracle, 2016. [Citação: 27 de 11 de 2016.] https://netbeans.org/.

16. Differences between NetBeans Platform and Eclipse RCP. [Online] NetBeans. [Citação: 5 de 12 de 2016.] https://netbeans.org/features/platform/compare.html.

17. **Sterling, Leon e Shapiro, Ehud.** *The Art of Prolog.* s.l. : The MIT Press, 1999.

18. **Clocksin, F., William e Mellish, S., Christopher.** *Programming in Prolog.* s.l. : Springer, 2003.

19. **AB, SICS Swedish ICT.** SICStus Prolog Customer References. [Online] [Citação: 2016 de 12 de 5.] https://sicstus.sics.se/customers.html.

20. **Dias, Artur Miguel.** CxProlog Page. [Online] 02 de 03 de 1993. [Citação: 16 de 08 de 2016.] http://ctp.di.fct.unl.pt/~amd/cxprolog/.

21. **Bonn, University of.** The Prolog Development Tool - A Prolog IDE for Eclipse. [Online] University of Bonn, 17 de 04 de 2016. [Citação: 27 de 11 de 2016.] http://sewiki.iai.uni-bonn.de/research/pdt/docs/start.

22. SICStus Prolog IDE (SPIDER). [Online] 28 de 11 de 2016. [Citação: 29 de 11 de 2016.] https://sicstus.sics.se/spider/.

23. **Calejo, Dr. Miguel.** Prolog Studio. [Online] InterProlog Consulting. [Citação: 28 de 11 de 2016.] http://interprolog.com/interprolog-studio/.

24. **Carvalho, Igor Emanuel Viera de.** *Plugin de CxProlog para o CodeBlocks IDE.* Lisboa : s.n., 2009.

25. *How Much Integrated Development Environments (IDEs) Improve Productivity?* **Zayour, Iyad e Hajjdiab, Hassan.** 10, 2013, JOURNAL OF SOFTWARE, Vol. 8, pp. 24-31.

26. **Fowler, Martin e Beck, Kent.** *Refactoring: Improving the Design of Existing Code.* s.l. : Addison-Wesley, 1999.

27. **Fowler, Martin.** Catalog of Refactorings. [Online] 10 de 12 de 2013. [Citação: 27 de 12 de 2015.] http://refactoring.com/catalog/index.html.

28. What is refactoring? Benefits of a code refactoring. [Online] DevExpress, 7 de 11 de 2010. [Citação: 27 de 12 de 2015.] http://goo.gl/uwO6AC.

29. **Eclipse.** Eclipse documentation - Current Release. [Online] [Citação: 20 de 09 de 2016.] http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Feditors_highlighting.html.

30. **Deva, Prashant.** Folding in Eclipse Text Editors. [Online] 11 de 03 de 2005. [Citação: 20 de 09 de 2016.] https://eclipse.org/articles/Article-Folding-in-Eclipse-Text-Editors/folding.html.

31. —. Create a commercial-quality Eclipse IDE, Part 2: The user interface. [Online] Placid Systems, 17 de October de 2006. [Citação: 28 de 11 de 2016.] https://www.ibm.com/developerworks/opensource/tutorials/os-ecl-commplgin2/.

32. *Smalltalk-80.* [Online] [Citação: 27 de 12 de 2015.] http://goo.gl/Qi7AnB.

33. **Burbeck, Steve.** *Applications Programming in Smalltalk-80.* 2012.

34. **Skerrett, Ian.** Eclipse Community Survey Results for 2013. [Online] [Citação: 27 de 12 de 2015.] http://wp.me/p1HAa-H9.

35. *CoRED: browser-based Collaborative Real-time Editor for Java web applications.* **Lautamäki, Janne, et al., et al.** 2012. CSCW '12.

36. *Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code.* **J.Brandt, et al., et al.** 2009. SIGCHI.

37. **Bruch, Marcel, et al., et al.** IDE 2.0: Collective Intelligence in Software Development. 2010.

38. **Dubois, Catherine, Giannakopoulou, Dimitra e Méry, Dominique.** 2014. 1st Workshop on Formal Integrated Development Environment.

39. *The Dafny Integrated Development Environment.* **Leino, K. Rustan M. e Wüstholz, Valentin.** 2014. 1st Workshop on Formal Integrated Development Environment.

40. **Carbonnelle, Pierre.** Top IDE Index. [Online] 2016. [Citação: 27 de 11 de 2016.] https://pypl.github.io/IDE.html.

41. **Cancinos, Claudio.** Prolog Development Tools - ProDT. [Online] 20 de 02 de 2012. [Citação: 27 de 11 de 2016.] http://prodevtools.sourceforge.net/index.html.

12 ANEXOS

12.1 Anexo 1 - Arquitetura da implementação do PDT

Um IDE desenvolvido na plataforma Eclipse pode ser constituído por vários plugins. Vejamos o exemplo da arquitetura do PDT (figura 55).

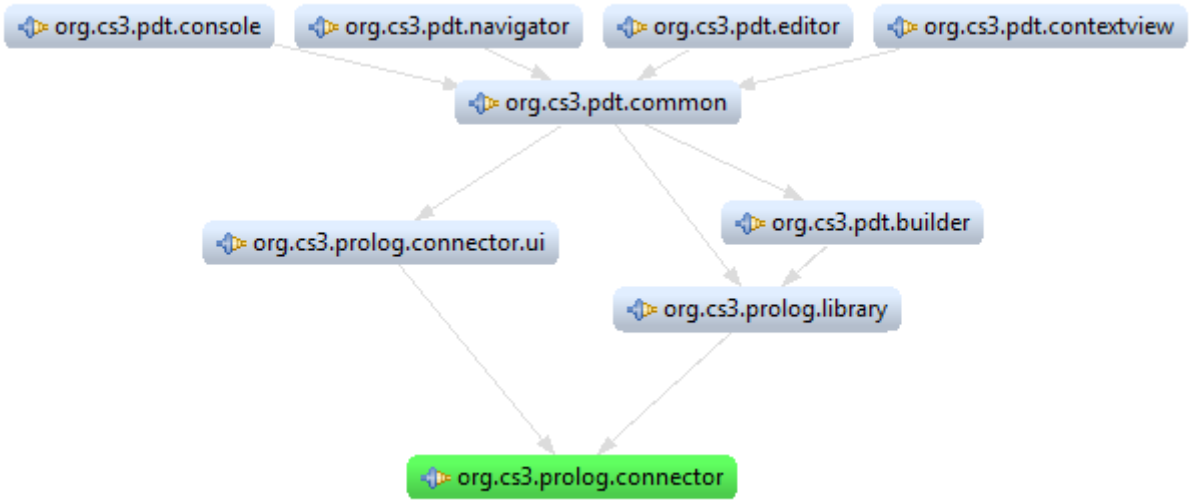


Figura 62 – Arquitetura do PDT

A imagem acima apresentada, revela que o PDT é constituído por nove *plugins*.

Cada um desse *plugins* tem uma função específica, sendo que alguns *plugins* podem estar dependentes de outros.

Por exemplo o *plugin* a verde **org.cs3.prolog.connector** é responsável por estabelecer a conexão entre processos Java/Eclipse e Prolog. Esse *plugin* é vital para o funcionamento do PDT pois todos os outros *plugins* que formam o PDT estão dependentes dele, direta ou indiretamente. No entanto, o conector pode ser usado independentemente do resto do PDT. Ou seja, é possível realizar a conexão entre Prolog e Java sem a existência do editor ou consola do PDT. Por outro lado, tanto editor como consola necessitam do conector para funcionar. A consola obviamente necessita de comunicar com o Prolog e o editor também necessita dessa comunicação para pedir informações para as suas funcionalidades como quais os termos *builtin* para o realce de sintaxe.

Dividir em vários *plugins* um projeto na plataforma Eclipse acarreta vantagens como:

- ✓ **Simplicidade** - Cada *plugin* tem tipicamente uma função, assim os programadores têm um único foco.
- ✓ **Desenvolvimento paralelo** - Algumas funcionalidades podem ser implementadas como componentes separados, assim podem ser desenvolvidas paralelamente por diferentes equipas.

E desvantagens como:

- **Dificuldade de testar** - Apesar de cada *plugin* poder funcionar quando testado individualmente, interações entre *plugins* podem causar novos problemas, inclusive alguns *bugs* podem só aparecer com certas combinações de *plugins*.
- **Separação artificial** - Um *plugin* tipicamente tem um único foco. Mas nem sempre é fácil definir qual esse foco e se manter leal ao mesmo. Ao longo do desenvolvimento dum *plugin* os objetivos podem mudar, o que podem levar à necessidade de separar o *plugin* em dois ou até convergir o mesmo com outro *plugin*.

12.2 Anexo 2 - Manual de utilização

Nesta secção são dadas algumas dicas para auxiliar o utilizador do CxIDE a obter o máximo proveito de todas as funcionalidades disponíveis.

12.2.1 Perspetiva do CxIDE

Para trabalhar no CxIDE é aconselhada a utilização da **CxPerspective**. Para mudar para a perspetiva do CxProlog (figura 63), no Eclipse, basta ir ao menu:

Window → Open Perspective → Other → CxPerspective

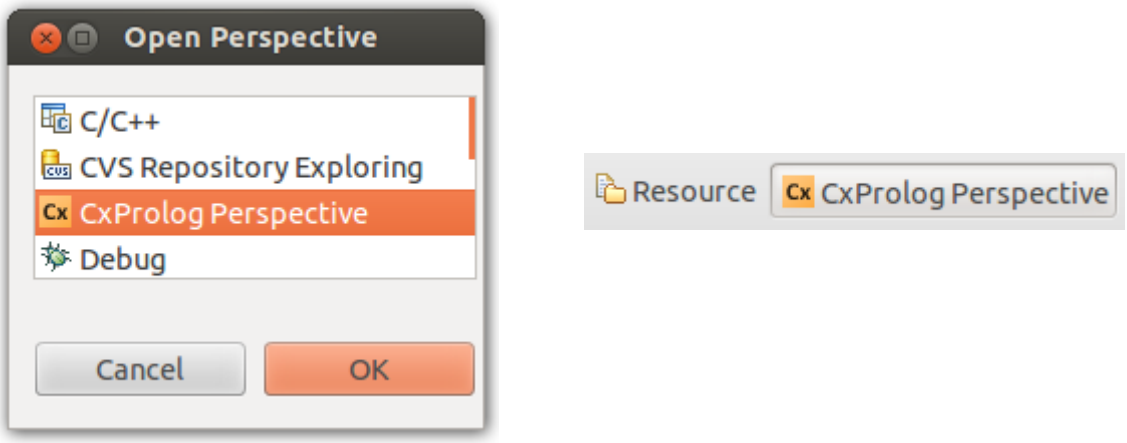


Figura 63 – Seleção da perspetiva CxProlog

A perspetiva do CxProlog (figura 64) engloba todas as vistas relacionados com o CxIDE. Essas vistas são:

Navegador de Projetos – O navegador padrão utilizado no Eclipse.

CxEditor – Edição de ficheiros CxProlog. Disponibiliza várias funcionalidades que beneficiam o utilizador.

Consola – Consola dinâmica que executa sobre a biblioteca dinâmica ou consola que opera sobre uma instalação local.

Vista estruturada – Representação visual do conteúdo em edição no CxEditor.

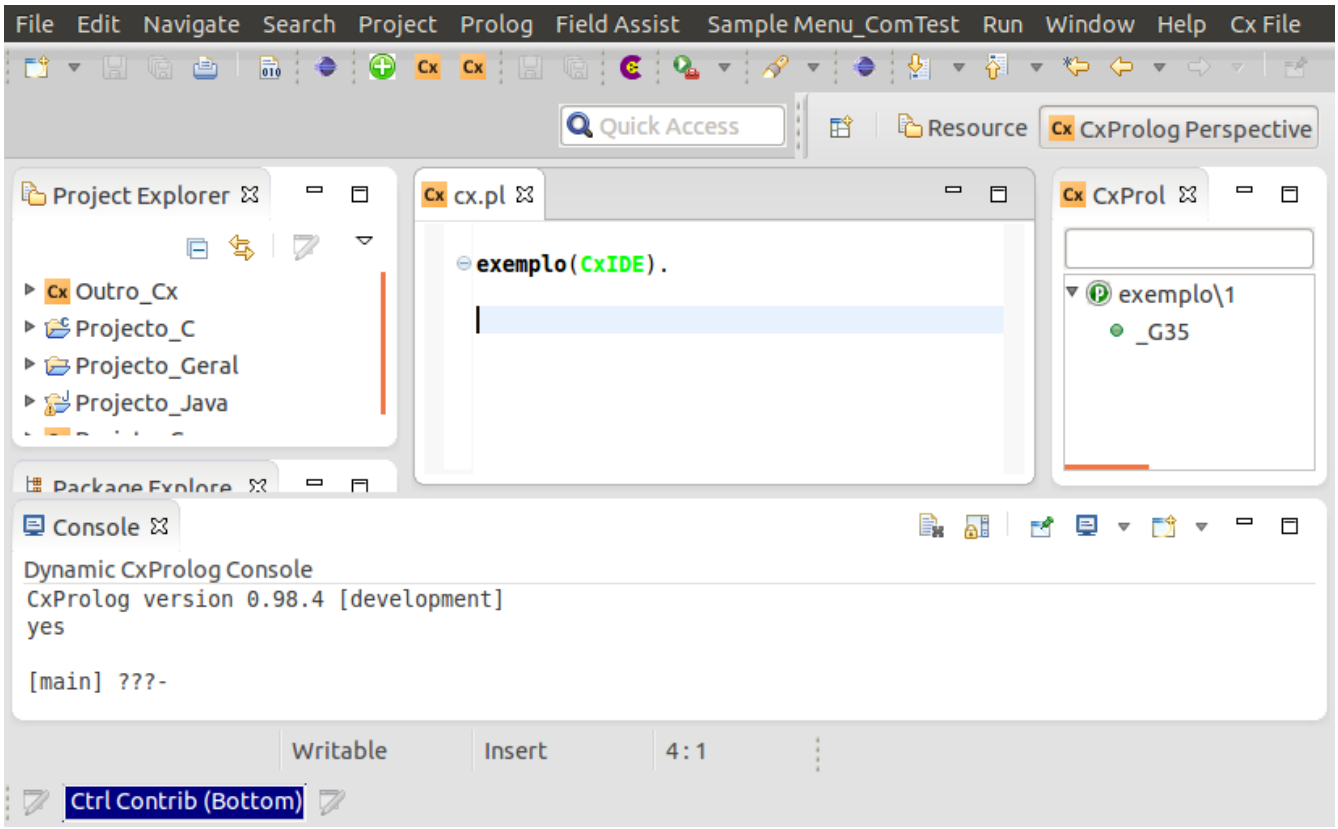


Figura 64 – Perspetiva do CxIDE no Eclipse

12.2.2 Navegador de projetos

Existe a noção de projeto CxProlog (**CxProject**), o que permite uma fácil identificação dos projetos do CxIDE (figura 65).

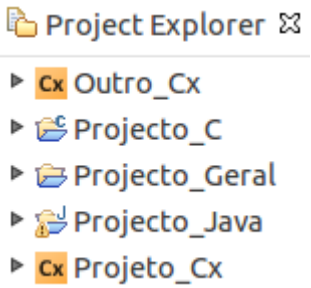


Figura 65 – Navegador de projetos

Os CxProject são representados com o ícone Cx

12.2.3 Criar projeto CxProlog

Para criar um CxProject basta seguir os seguintes passos:

File →New → Other → CxProlog Project

Após os mesmos, será exibido um *wizard* para a criação dum novo projeto CxProlog (figura 66).

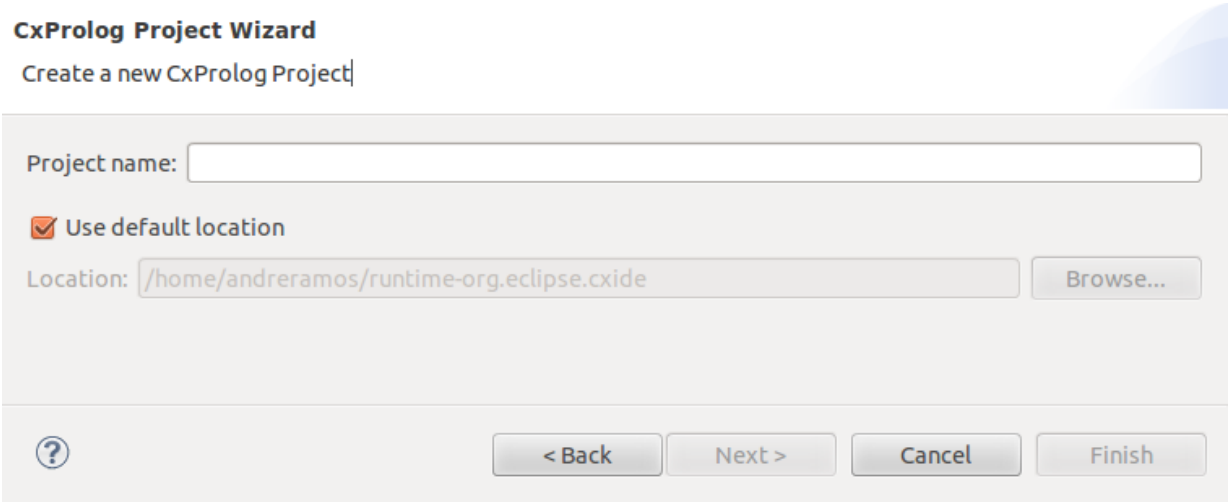


Figura 66 – Wizard para a criação de projetos CxProlog

12.2.4 Converter um projeto

É possível converter um projeto duma diferente natureza para um projeto CxProlog.

Para tal, deve ser utilizado o menu de contexto dos projetos (clique direito sobre um projeto) e selecionar **Configure → Convert to CxProject** (figura 67). Após esta operação o projeto convertido será um projeto CxProlog.

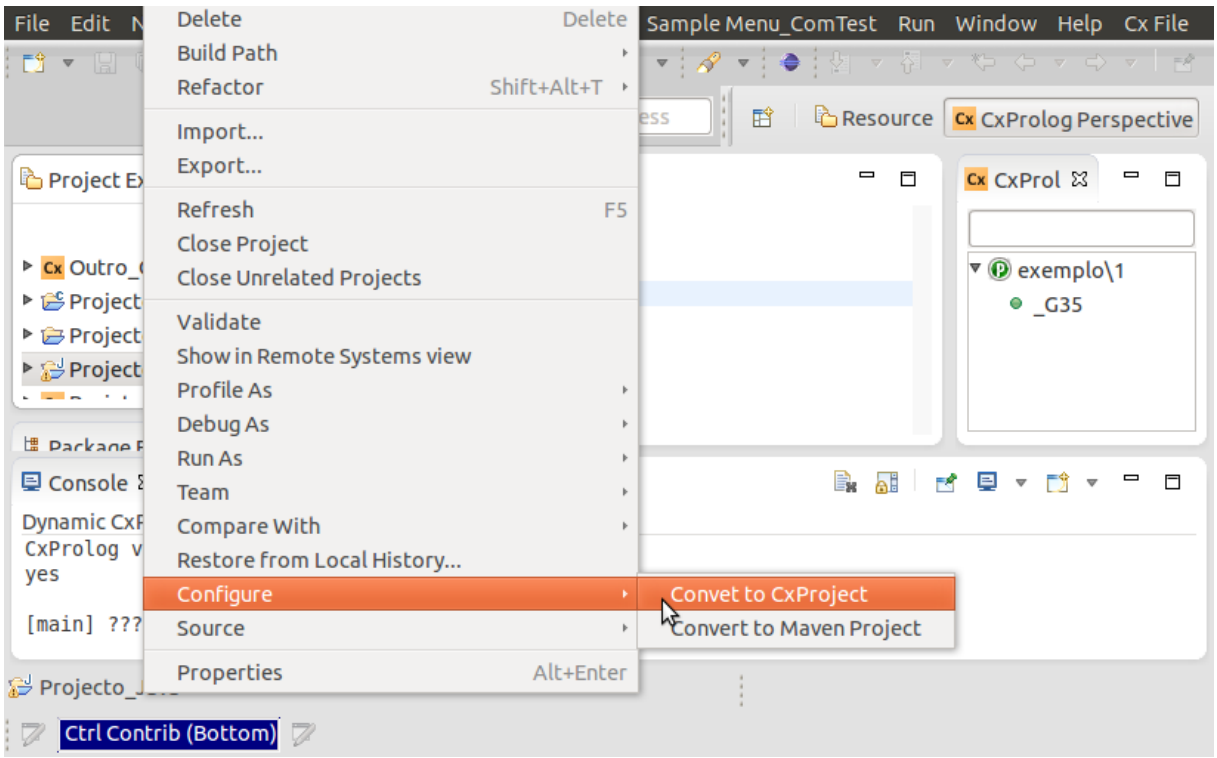


Figura 67 – Exemplo de conversão dum projeto

12.2.5 CxEditor

O CxEditor é o editor do CxIDE que foi desenvolvido para agilizar a edição de ficheiros CxProlog.

São suportadas as seguintes funcionalidades:

- Realce de sintaxe
- Validação sintática
- Realce de erros
- Expansão e colapso

O CxEditor é aberto automaticamente para edição de ficheiros com a extensão *pl*. Caso o mesmo não aconteça é sinal que o Eclipse está configurado para usar um outro editor para este tipo de ficheiros. Isto acontece quando existem outros *plugins* para a programação de Prolog instalados ou o utilizador tenha alterado as preferências dos editores.

Para abrir um ficheiro com o CxEditor, devem ser seguidos os seguintes passos:

Menu contexto dos ficheiros → Open With → CxEditor

Para definir o CxEditor como padrão para a edição de ficheiro *pl*.

Window → Preferences → General → Editors → File Associations

Na página de preferências **File Associations** o utilizador poderá associar editores a determinadas extensões. Sendo igualmente possível definir qual o editor por omissão para uma determinada extensão. Neste caso, o utilizador deverá associar o CxEditor à extensão *pl* e caso existam outros editores associados o CxEditor deve ser definido como **Default** (figura 68).

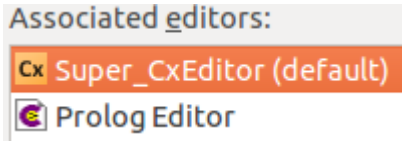


Figura 68 – Definição do CxEditor como editor por omissão na página File Associations

12.2.6 Realce de sintaxe

Por omissão, o CxEditor suporta realce de sintaxe. Nesse realce de sintaxe é realizada a distinção dos seguintes termos:

- Variáveis
- Texto
- Predicados *builtin*
- Predicados *user defined*
- Comentários
- Strings

Para modificar a coloração do realce de sintaxe o utilizador deve usar a página de preferências do CxEditor:

Window → Preferences → CxProlog → CxEditor

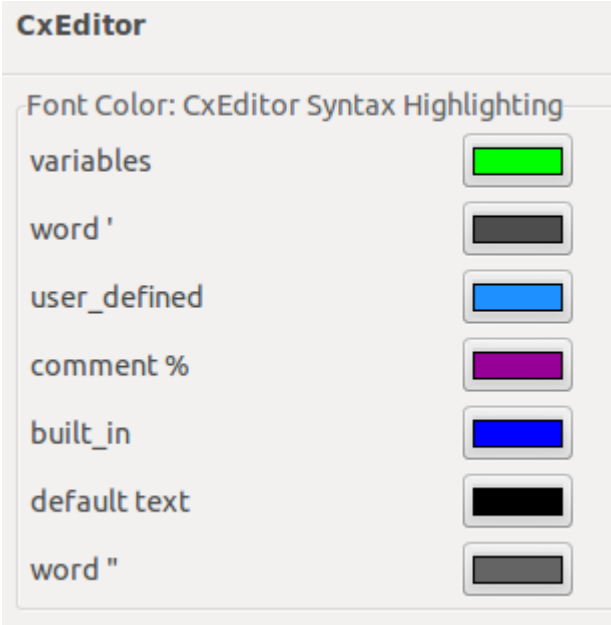


Figura 69 – Preferências para alteração do realce de sintaxe

12.2.7 Configuração

O utilizador têm a possibilidade de definir o seu próprio realce de sintaxe. Para definir o realce de sintaxe podem ser usados os seguintes predicados:

```
editor_singleLineRule(FieldName, StartSeq, EndSeq, EscChar, R, G, B)
editor_endLineRule(FieldName, StartSeq,R,G,B)
editor_varRule(FieldName,R,G,B)
editor_addNormalTextRule(FieldName,R,G,B)
editor_addWordRule(FieldName,Word,R,G,B)
editor_addWordsRule(FieldName,[Head|Tail],R, G, B)
editor_addPropertyHighlight(Pro,R,G,B)
```

Cada utilização dos predicados acima descritos adicionará um novo campo na página de preferências do CxEditor. Assim é fácil modificar a coloração do realce de sintaxe em execução.

O utilizador deverá definir o realce de sintaxe editado o ficheiro *root.pl* ou usando a consola dinâmica do CxIDE.

Por defeito, o CxIDE define o seu realce de sintaxe de acordo com as seguintes regras que estão definidas no *root.pl*:

```
%Definir regras para syntax highlight NULL POINTER EXCEPTION
%Definir texto entre " como um certo highlight
:-editor_singleLineRule('word "','"', '"', '\\', 100, 100, 100).
:-editor_singleLineRule('word ''','''', '''', '\\', 50, 50, 50).
:-editor_endLineRule('comment %','%', 150, 0, 150).
:-editor_varRule('variables',0,255,0).
:-editor_addNormalTextRule('default text',0,0,0).
:-editor_addWordRule('special','andre',200,200,0).
:-editor_addPropertyHighlight('built_in',0,0,255).
:-editor_addPropertyHighlight('user_defined',0,255,0).
```

Obviamente o utilizador pode eliminar, adicionar ou modificar tais regras editando o *root.pl*

12.2.8 Completação automática

Tal como em outros IDEs existentes para Eclipse, o utilizador necessita de usar **CTRL+SPACE** para ser revelada a lista de propostas de completação.

De forma mais detalhada, durante a edição dum ficheiro no CxEditor o utilizador poderá premir **CTRL+SPACE** obtendo assim uma lista de possíveis completações. Essa lista inclui as assinaturas predicados *builtin* e *user definied* do CxProlog, tal como, uma descrição se a mesma existir (figura 70).

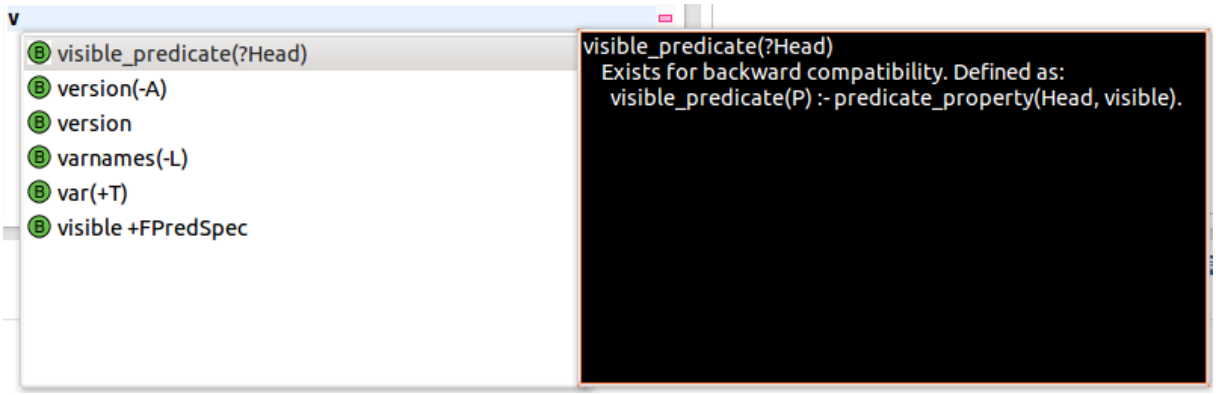


Figura 70 – Completação automática do CxIDE

O utilizador poderá selecionar um dos elementos dessa lista e o mesmo aparecerá automaticamente no ficheiro em edição.

A completação automática está dependente de um formato *xml* do manual do CxProlog.

O utilizador poderá escolher um ficheiro *xml*, com a mesma estrutura que o inicial, para definir uma nova completação automática. Para tal existe o predicado CxProlog:

```
editor_setContentAssistFile(FilePath)
```

```
<pred>
        <id>forall(+G1,+G2)</id>
        <desc>forall(+G1, +G2)
    Checks whether the goal G2 can be proved for all the alternative
    bindings of a goal G1. Defined as:
    *forall(G1, G2) :- \+ (G1, \+ G2).*</desc>
</pred>
```

O editor também disponibiliza colapso e expansão a sua utilização é intuitiva.

12.2.9 Menu de contexto

Para além dos normais itens dos menus de contexto dum editor Eclipse (exs: **Cut**, **Copy**) o menu de contexto do **CxEditor** inclui o item CxProlog. Este item engloba várias subitens que representam operações especificas relacionadas com o CxIDE. Uma dessas operações é a injeção de golos selecionados no editor para a consola dinâmica. Pode ser usado o atalho **CTRL+F2** para o mesmo efeito.

O utilizador pode adicionar novos subitens ao item CxProlog. Para tal é oferecido o seguinte predicado:

```
add_contextItem(Nome,Comando)
```

12.2.10 Consultar ficheiro em edição

Para consultar ficheiros em edição no **CxEditor** o utilizador poderá usar os botões da barra de ferramentas com o ícone do CxIDE (figura 71) ou utilizar os atalhos de teclado.

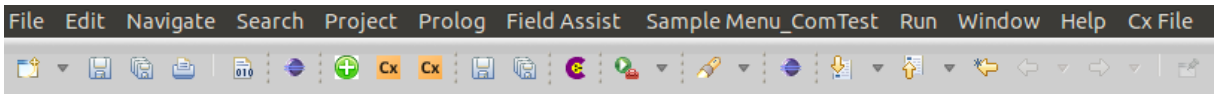


Figura 71 – Barra de menus e ferramentas do Eclipse com o plugin CxIDE

1º Botão consulta o atual ficheiro em edição. Premir **CTRL+F3** para o mesmo efeito.

2º Botão consulta todos os ficheiros CxProlog em abertos no CxEditor. Premir **CTRL+F4** para o mesmo efeito.

12.2.11 Consola

O CxIDE disponibiliza duas consolas:

CxProlog Internal Console – Principal consola do CxIDE. Funciona sobre a biblioteca dinâmica do CxProlog, não sendo necessária uma instalação local do CxProlog. Só nesta consola podem ser executados os predicados específicos à extensão ou configuração do CxIDE.

CxProlog External Console – Consola secundária do CxIDE. Deverá apenas ser utilizada caso o utilizador pretender usar uma instalação local do CxProlog.

Para mudar de consola basta ir à vista **Console** e selecionar a consola pretendida no separador:

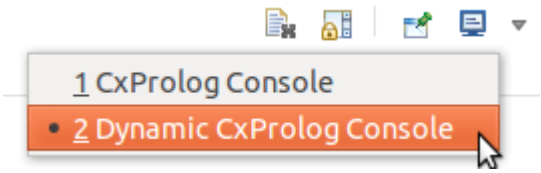


Figura 72 – Seleção das consolas do CxIDE

Para mudar a instalação local de CxProlog utilizada na consola ou para modificar as definições de conexão do servidor devem ser utilizadas as preferências:

Window → Preferences → CxProlog

12.2.12 Preferências

A preferências do CxEditor podem ser encontradas em:

Window → Preferences → CxProlog

Existem duas páginas de preferências:

CxProlog

CxProlog Path: Caminho para a instalação local de CxProlog. Esta instalação é utilizada para a consola secundária do CxIDE.

- Server Ip - Ip do computador a conectar a consola secundária do CxIDE
- Server Port - Porta do computador a conectar a consola secundária do CxIDE

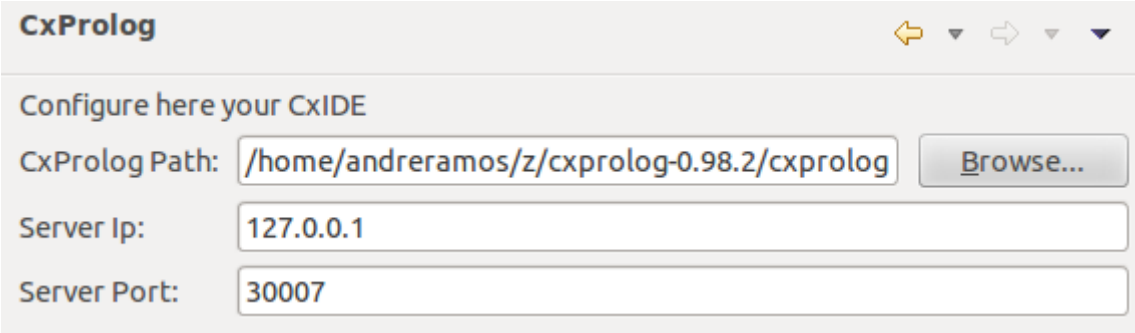


Figura 73 – Página de preferências do CxIDE

CxEditor

Nesta página de preferências (figura 74) o utilizador pode alterar a coloração das regras de sintaxe do CxEditor.

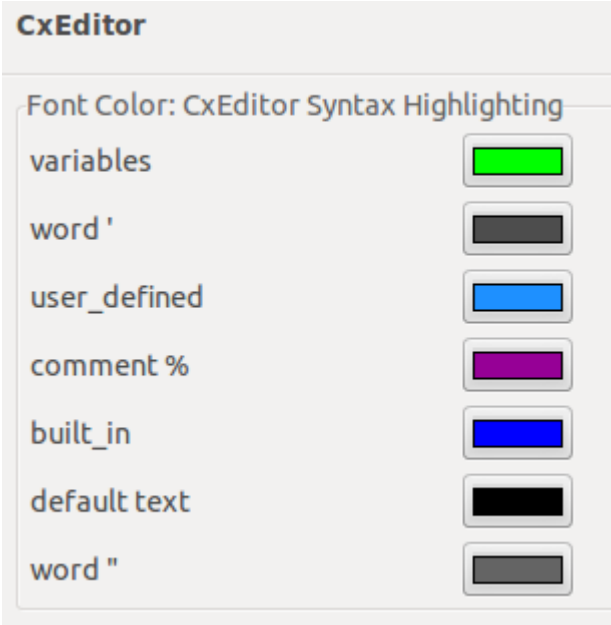


Figura 74 – Página de preferências do CxEditor

12.2.13 Vista Estruturada

Apresenta a estrutura do ficheiro em edição. Um duplo clique num dos elementos da vista estruturada levará o CxEditor a selecionar e exibir a declaração do elemento no ficheiro em edição (figura 75).

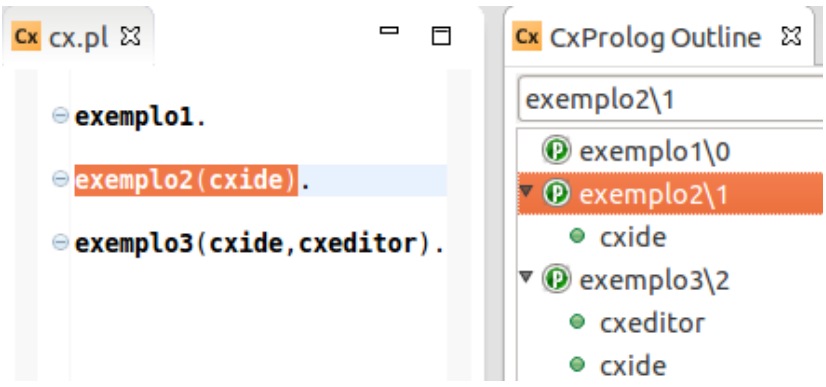


Figura 75 – Vista estrutrada (CxOutline) do CxIDE

12.2.14 Outros

Durante a implementação do CxIDE foi necessário desenvolver predicados que facilitam operações necessárias entre CxProlog e Java. O utilizador dispõe de alguns desses predicados:

```
editor_getCurrOpenFilePath(Path)
editor_getAllOpenFilePaths(FilePaths)
editor_getCurrentContent(Content)
editor_getCodeErrors
editor_injectSelCode
selectedText(Text)
open_file(Path)
```

12.2.15 Estender o CxIDE

O utilizador pode estender ou configurar o CxIDE usando CxProlog. Para isso poderá editar o ficheiro *config.pl*, para que o CxIDE inicie com as novas definições. Enquanto o ficheiro *config.pl* não for novamente editado pelo utilizador as extensões ou configurações serão persistentes.

O utilizador pode igualmente estender ou configurar o CxIDE durante a execução do mesmo, usando a consola dinâmica.

No anexo 3 é apresentada a lista de todos os predicados CxProlog especificamente desenvolvidos para facilitar ao utilizador a extensão ou configuração do CxIDE.

12.3 Anexo 3 - Folha de dicas

Menus	`create_menu(NomeMenu)`	Adiciona um novo menu, caso não exista, com o nome **NomeMenu** à barra principal de menus do Eclipse. Caso já exista um menu **NomeMenu** a adição não será realizada e o utilizador será notificado.
	`create_menu(NomeMenu, Item, CMD)`	Adiciona um novo item com o nome **Item** ao menu com o **NomeMenu**. O menu **NomeMenu** será criado caso não exista. O comando CxProlog **CMD** define o comportamento do item, isto é, quando o utilizador selecionar esse item será executado o comando dado.
	`create_menu(NomeMenu,Items)`	Adiciona um novo menu, caso não exista, com o nome **NomeMenu**. A esse menu serão adicionados os itens e respetivos comportamentos descritos na lista **Items**.
	`delete_menu(NomeMenu)`	Elimina um menu com o nome **NomeMenu**.
	`listMenus`	Revela um diálogo com a lista de todos os menus criados através do CxProlog.
Diálogos	`getTextField(Dialog,Field_Label,Value)`	Obtém o valor textual na variável **Value** do campo de inserção com a etiqueta **Label** da janela de diálogo **Dialog**.
	`getNumField(Dialog,Field_Label,N)`	Obtém o valor numérico na variável **N** do campo de inserção com a etiqueta **Label** da janela de diálogo **Dialog**.

	`dialog_setAction(Dialog, Action)`	Define o comportamento da janela de diálogo **Dialog** como **Action**.
	`file_chooser(FilePath)`	Abre uma janela de diálogo para a navegação nas diretorias com o propósito de selecionar um ficheiro. O caminho absoluto do ficheiro selecionado é retornado na variável **FilePath**.
	`dir_chooser(Value)`	Abre uma janela de diálogo para a navegação nas diretorias com o propósito de selecionar uma diretoria. O caminho absoluto da diretoria selecionado é retornado na variável **Value**.
	`fd_chooser(Value)`	Abre uma janela de diálogo para a navegação nas diretorias com o propósito de selecionar uma diretoria/ficheiro. O caminho absoluto da diretoria/ficheiro selecionado é retornado na variável **Value**.

Tabela 9 – Folha de dicas do CxIDE 1

Editor	`editor_singleLineRule(FieldName, StartSeq, EndSeq, EscChar, R, G, B)`	Predicado para a definição duma regra do realce de sintaxe. Esta regra identifica linhas que comecem com a sequência de carateres **StarSeq** e terminem com a sequência de carateres **EndSeq**. Sendo possível utilizar o caráter de escape **EscChar** nas sequências anteriores. A coloração das linhas limitadas por essas sequências é a definida por **R**, **G**, **B**.
	`editor_endLineRule(FieldName, StartSeq,R,G,B)`	Predicado para a definição duma regra do realce de sintaxe. Esta regra identifica linhas que comecem com a sequência de carateres **StartSeq**. A coloração das linhas limitadas por essa sequência é a definida por **R**, **G**, **B**.
	`editor_varRule(FieldName, R,G,B)`	Predicado para definição duma regra do realce de sintaxe. Esta regra define variáveis do CxProlog. A coloração das variáveis será a definida por **R**, **G**, **B**.
	`editor_addNormalTextRule(FieldName, R,G,B)`	Predicado para a definição duma regra do realce de sintaxe. Esta regra define a coloração do texto que não se adequa a nenhuma das outras regras do realce de sintaxe. A coloração desse texto é dada por **R**, **G**, **B**.

Tabela 10 – Folha de dicas do CxIDE 2

Editor	`editor_addWordRule(FieldName, Word, R,G,B)`	Predicado para a definição duma regra do realce de sintaxe. Esta regra define a coloração de palavras fixas **Word**. A coloração é definida por **R**, **G**, **B**.
	`editor_addWordsRule(FieldName, [Word1|WordN],R, G, B)`	Predicado para a definição duma regra de realce de sintaxe do **CxEditor**. Esta regra define a coloração de palavras fixas definidas na **Lista [Word1|WordN]**. A coloração é definida por **R**, **G**, **B** que podem variar entre 0-255.
	`editor_addPropertyHighlight(Pro, R,G,B)`	Predicado para a definição duma regra de realce de sintaxe do **CxEditor**. Esta regra define a coloração de predicados CxProlog com a propriedade **Pro**. Exemplos de propriedades: *built_in*, *user_defined*. Coloração definida por **R**, **G**, **B** que podem variar entre 0-255
	`editor_getCurrOpenFilePath(Path)`	Obtém o caminho absoluto do ficheiro atualmente em edição. O caminho é dado na variável **Path**.
	`editor_getAllOpenFilePaths(FilePaths)`	Obtém os caminhos absolutos de todos os ficheiros abertos no editor. Os caminhos são retornados na lista **FilePaths**.

Tabela 11 – Folha de dicas do CxIDE 3

Editor	**editor_getCurrentContent(Content)**	Obtém o conteúdo em edição. O conteúdo é retornado na variável **Content**.
	editor_getCodeErrors	Atualiza o realce de erros do ficheiro em edição no **CxEditor**.
	editor_injectSelCode	Injeta na consola do CxIDE o texto selecionado no **CxEditor**.
	editor_setContentAssistFile(FilePath)	Define um novo ficheiro que suporta a completação automática do **CxEditor**.
	add_contextItem(MenuName,CMD)	Adiciona um novo item ao menu de contexto CxProlog do **CxEditor**, O novo item terá o título **MenuName** e o seu comportamento é definido por CMD.
	open_file(Path)	Abre no **CxEditor** o ficheiro cujo caminho absoluto é **Path**.
	selectedText(Text)	Obtém para a variável **Text** o texto atualmente selecionado no **CxEditor**.

Tabela 12 – Folha de dicas do CxIDE 4

12.4 Anexo 4 - Criação programaticamente de diálogos

```
  /**    Cria um novo diálogo com o título/id dado pelo utilizador
  * @param titulo - O título do diálogo*/
  public static void createDialog(String titulo){
  CxDialog dialog = new CxDialog(titulo);
  dialogs.put(titulo+"_ID",dialog);     }

  /** Adiciona um novo campo de texto a um diálogo existente
  * @param titulo - O título do diálogo
  * @param label_name - O nome da etiqueta do novo diálogo*/
  public static void addTextFiel_dialog(String titulo, String label_name){
  CxDialog dialog = dialogs.get(titulo+"_ID");
  dialog.addTextForm(label_name);
  }


  /**   Adiciona um novo campo numérico a um diálogo existente
  * @param titulo - O título do diálogo
  * @param label_name - O nome da etiqueta do novo diálogo
  */
  public static void addNumTextFiel_dialog(String titulo, String
label_name){
  CxDialog dialog = dialogs.get(titulo+"_ID");
  dialog.addNumForm(label_name);
  }

  /** Revela um diálogo
  * @param titulo - O título do diálogo a revelar*/
  public static void showDialog(String titulo){
  CxDialog dialog = dialogs.get(titulo+"_ID");
```

```
  dialog.open();
  }

  /** Obtém o valor dum campo textual dum diálogo existente
  * @param dialog_title - O título do diálogo
  * @param field - O nome da etiqueta do campo textual
  * @return - O valor atual do campo textual do diálogo selecionado*/
  public static String dialog_getTextField(String dialog_title, String
field){
  CxDialog dialog = dialogs.get(dialog_title+"_ID");
  return dialog.getTextField(field);
  }

  /**Obtém o valor dum campo numérico de um diálogo existente
  * @param dialog_title - O título do diálogo
  * @param field - O nome da etiqueta do campo numérico
  * @return o valor atual do campo textual do diálogo selecionado*/
  public static String dialog_getNumField(String dialog_title, String
field){
  CxDialog dialog = dialogs.get(dialog_title+"_ID");
  return String.valueOf(dialog.getNumField(field))
  }

  /** Modifica o comportamento dum diálogo existente
  * O utilizador terá de definir um novo comportamento do diálogo
recorrenco ao CxProlog
  * @param dialog_id - O identificador do diálogo
  * @param action_cmd - Os predicados que definem o novo comportamento do
diálogo
  */
  public static void dialog_setAction(String dialog_title, String
action_cmd){
  CxDialog dialog = dialogs.get(dialog_title+"_ID");
  CxDialog.setAction(action_cmd);
  }
```

12.5 Anexo 5 – Ficheiro root.pl

Eis o conteúdo do ficheiro *root.pl* que define predicados que podem ser usados no CxIDE:

```
%Criar menu com o título "NomeMenu"

create_menu(NomeMenu):-
java_call('org/eclipse/cxide/Menu_ops/Menu_Operations','createMenu:(Ljava/lang/String;)V',[Nome
Menu],_).

%Criar menu com o título "NomeMenu" e com os itens definidos na "Lista"

create_menu(NomeMenu,Lista):-java_call('org/eclipse/cxide/Menu_ops/Menu_Operations',
'createMenu:(Ljava/lang/String;[Ljava/lang/String;)V', [NomeMenu,Lista], R).

%Criar menu com o título "NomeMenu" com um único item com nome "Item_Name" e que executa a
%acao "Action"

create_menu(NomeMenu,Item_Name,Action):-
java_call('org/eclipse/cxide/Menu_ops/Menu_Operations','createMenu:(Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;)V',[NomeMenu,Item_Name,Action],_).

%Elimina o menu com o id "Menu_ID", se for um menu criado pelo utilizador todos os seus sub-
%menus também sao eliminados. Caso contrário os sub-menus continuarão a existir

delete_menu(Menu_ID):-
java_call('org/eclipse/cxide/Menu_ops/Menu_Operations','deleteMenu:(Ljava/lang/String;)V',[Menu_I
D],_).

%Criar e guarda uma janela de diálogo com o título "Titulo" sem output

create_dialog(Titulo):-
java_call('org/eclipse/cxide/Menu_ops/Dialog_Operations','createDialog:(Ljava/lang/String;)V',[Titul
o],_).

%Exibe uma janela de diálogo com o título "Titulo" que tenha sido posteriormente criada pelo
%utilizador.

show_dialog(Titulo):-
java_call('org/eclipse/cxide/Menu_ops/Dialog_Operations','showDialog:(Ljava/lang/String;)V',[Titulo
],_).

%Exibe uma janela de diálogo com a lista de todos os menus criados via CxProlog

listMenus:-
java_call('org/eclipse/cxide/Menu_ops/Menu_Operations','printMenusCreated:()V',[],_).

%Adiciona um campo de texto com a Label ao dialog Titulo

add_textForm_dialog(Titulo,Label):-
java_call('org/eclipse/cxide/Menu_ops/Dialog_Operations','addTextFiel_dialog:(Ljava/lang/String;Lja
va/lang/String;)V',[Titulo,Label],_).

%Adiciona um campo de numero com a Label ao dialog Titulo

add_numForm_dialog(Titulo,Label):-
java_call('org/eclipse/cxide/Menu_ops/Dialog_Operations','addNumTextFiel_dialog:(Ljava/lang/Strin
g;Ljava/lang/String;)V',[Titulo,Label],_).

%Obtém na variável Value o conteúdo do campo de edição de texto com a etiqueta Field_Label do
%dialog com título Dialog

getTextField(Dialog,Field_Label,Value):-
java_call('org/eclipse/cxide/Menu_ops/Dialog_Operations','dialog_getTextField:(Ljava/lang/String;Lj
ava/lang/String;)Ljava/lang/String;',[Dialog,Field_Label],Value),writeln(Value).
```

%Obtém na variável Value o conteúdo do campo de edição de numéricos com a etiqueta Field_Label %do dialog com título Dialog

```
getNumField(Dialog,Field_Label,N):-
java_call('org/eclipse/cxide/Menu_ops/Dialog_Operations','dialog_getNumField:(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;',[Dialog,Field_Label],V),atom_codes(V,L),number_codes(N,L),writeln(N).
```

%Cria um diálogo que revelará todos os elementos da Lista (2º parâmetro).

```
dialog_setOutput(Dialog,[Head|Tail]):-
java_call('org/eclipse/cxide/Menu_ops/Dialog_Operations','setOutput:(Ljava/lang/String;[Ljava/lang/Object;)V',[Dialog,[Head|Tail]],_).
```

%Define o comportamento do botão OK do diálogo com título Dialog para executar o golo CxProlog %Action

```
dialog_setAction(Dialog,Action):-
java_call('org/eclipse/cxide/Menu_ops/Dialog_Operations','dialog_setAction:(Ljava/lang/String;Ljava/lang/String;)V',[Dialog,Action],_).
```

%Exibe um diálogo para navegar em diretórias com o propósito de seleção dum ficheiro. O caminho %absoluto do ficheiro selecionado será o valor de FilePath

```
    file_chooser(FilePath):-
java_call('org/eclipse/cxide/Menu_ops/FileChooser_Operations','FileChooseDialog:()Ljava/lang/String;',[],FilePath).
```

%Exibe um diálogo para navegar em diretórias com o propósito de seleção dum diretório. O caminho %absoluto do ficheiro selecionado será o valor de FilePath

```
    dir_chooser(Value):-
java_call('org/eclipse/cxide/Menu_ops/FileChooser_Operations','DirChooseDialog:()Ljava/lang/String;',[],Value),writeln(Value).
```

%Exibe um diálogo para navegar em diretórias com o propósito de seleção dum ficheiro ou diretório. %O caminho %absoluto do ficheiro selecionado será o valor de FilePath

```
    fd_chooser(Value):-
java_call('org/eclipse/cxide/Menu_ops/FileChooser_Operations','FileDirChooseDialog:()Ljava/lang/String;',[],Value),writeln(Value).
```

% Abrir ficheiro externo no CxEditor

```
    open_file(Path):-
java_call('org/eclipse/cxide/Menu_ops/FileChooser_Operations','open_external_file_onEditor:(Ljava/lang/String;)V',[Path],_).
```

% Abre automaticamento no CxEditor o ficheiro config.pl

```
config:-java_call('org/eclipse/cxide/Menu_ops/FileChooser_Operations','openConfig:()V',[],_).
```

%Adiciona novos itens ao menu de contexto do CxEditor

```
    add_contextItem(MenuName,CMD):-
java_call('org/eclipse/cxide/Menu_ops/EditorContextMenu','add_item_editor:(Ljava/lang/String;Ljava/lang/String;)V',[MenuName,CMD],_).
```

% Obtem na variável Text o texto atualmente selecionado no CxEditor

```
    selectedText(Text):-
java_call('org/eclipse/cxide/Menu_ops/Editor_Operations','getSelectedText:()Ljava/lang/String;',[],X),
java_convert('Ljava/lang/String;', X, Text),writeln(Text).
```

```
%Obtem a lista de predicados com a dada propriedade
    getPredicates(B,L):-findall(A,predicate_property(A,B),L).

%recebe uma lista de predicados e retorna a lista com os seus funtores
    getFunctors([],[]).
    getFunctors([X|Xs],[Fun|Ys]) :- functor(X, Fun, Ar), getFunctors(Xs,Ys).

%obtem a lista de funtores de todos os predicados com a propriedade Pro
    getPredicatesWithProperty(Pro,Pre):-getPredicates(Pro,L),getFunctors(L,Pre).

% (Inicio) Regras de definição de realce de sintaxe ----------------------------------------
    editor_singleLineRule(FieldName, StartSeq, EndSeq, EscChar, Red, Green, Blue):-
java_call('org/eclipse/cxide/utilities/Editor_Utilities','singleLineRule:(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;III)V',[FieldName,StartSeq, EndSeq, EscChar, Red, Green, Blue],_).

    editor_endLineRule(FieldName, StartSeq,Red,Green,Blue):-
java_call('org/eclipse/cxide/utilities/Editor_Utilities','endOfLineRule:(Ljava/lang/String;Ljava/lang/String;III)V',[FieldName, StartSeq, Red, Green, Blue],_).

    editor_varRule(FieldName,R,G,B):-
java_call('org/eclipse/cxide/utilities/Editor_Utilities','addVarRule:(Ljava/lang/String;III)V',[FieldName,R, G, B],_).

    editor_addNormalTextRule(FieldName,Red,Green,Blue):-
java_call('org/eclipse/cxide/utilities/Editor_Utilities','addDefaultTextRule:(Ljava/lang/String;III)V',[FieldName, Red, Green, Blue],_).

    editor_addWordRule(FieldName,Word,Red,Green,Blue):-
java_call('org/eclipse/cxide/utilities/Editor_Utilities','addWordRule:(Ljava/lang/String;Ljava/lang/String;III)V',[FieldName,Word,Red, Green, Blue],_).

    editor_addWordsRule(FieldName,[Head|Tail],R, G, B):-editor_addWordRule(FieldName,Head, R, G, B),editor_addWordsRule(FieldName,Tail, R, G, B).

    editor_addPropertyHighlight(Pro,R,G,B):-
getPredicatesWithProperty(Pro,List),getFunctors(List,Functors),editor_addWordsRule(Pro,Functors, R, G, B).
% (Fim) Regras de definição de realce de sintaxe ----------------------------------------
```

```
%Obtém na variável Path o caminho do ficheiro atualmente em edição no CxEditor

    editor_getCurrOpenFilePath(Path):-
java_call('org/eclipse/cxide/Menu_ops/Editor_Operations','getCurrentEditorFilePath:()Ljava/lang/String;',[],Path).

%Obtém na variável FilePaths os caminhos dos ficheiros atualmente em edição no CxEditor

    editor_getAllOpenFilePaths(FilePaths):-
java_call('org/eclipse/cxide/Menu_ops/Editor_Operations','getCurrentEditorAllFilesPath:()[Ljava/lang/String;',[], X), java_convert('[Ljava/lang/String;', X, FilePaths).

%Obtém na variável FilePaths os caminhos dos ficheiros atualmente em edição no CxEditor

    editor_getCurrentContent(Content):-
java_call('org/eclipse/cxide/Menu_ops/Editor_Operations','getCurrentEditorContent:()Ljava/lang/String;',[], Content).


%Analise sintática

    editor_getCodeErrors:-
editor_getCurrentContent(Code),code_summary(Code,Summary,Errors),java_call('org/eclipse/cxide/Menu_ops/Editor_Operations','errors_Folding_Outline:([Ljava/lang/Object;[Ljava/lang/Object;)V',[Summary,Errors],_).

%Injeta o texto selecionada do CxEditor na consola ativa do CxIDE

    editor_injectSelCode:-
java_call('org/eclipse/cxide/Menu_ops/Editor_Operations','injectSelCodeConsole:()V',[], _).


%Obter predicados builtin

    getBuiltins(Lista):-
getPredicatesWithProperty('built_in',Lista),writeln(Lista),java_call('org/eclipse/cxide/utilities/Editor_Utilities','setBuiltins:([Ljava/lang/String;)V',[Lista],_).

%Definir o ficheiro xml a utilizar na completação automática

    editor_setContentAssistFile(FilePath):-
java_call('org/eclipse/cxide/utilities/Editor_Utilities','setContentAssistFile:(Ljava/lang/String;)V',[FilePath],_).


%Usado para a vista estruturada

    getOutlineInfo([]):-true.

    getOutlineInfo([Head,Start,Offset|Tail]):-
atom_termq(Head,Term),functor(Term,Fun,Ar),extractArgs(Term,Ar,Args),java_call('org/eclipse/cxide/Menu_ops/Editor_Operations','addFileFunctor:(Ljava/lang/String;III)V',[Fun,Ar,Start,Offset],_),getOutlineInfo(Tail).

    getOutlineInfo([Head,Start,Offset|Tail]):-getOutlineInfo(Tail).


%Convert termos para a sua representação textual ou vice-versa

    atomsToTexts([],[]).

    atomsToTexts([Atom|Atoms],[Text|Texts]):-atom_term(Text,Atom),atomsToTexts(Atoms,Texts).
```

```
%Extrai os argumentos de um dado termo
    extractArgs(Term,Ar,List):-getArgs1(Term,
Ar,List),atomsToTexts(List,L2),java_call('org/eclipse/cxide/Menu_ops/Editor_Operations','addFileAr
g:([Ljava/lang/String;)V',[L2],_).
    getArgs1(Term, 0,[]).
    getArgs1(Term, Arity,L):-
    arg(Arity,Term,Arg),
        Ari is Arity-1,
```

12.6 Anexo 6 – Ficheiro de configurações (config.pl)

Eis o conteúdo inicial do ficheiro config.pl:

```
%Este ficheiro permite definir a configuração inicial do CxIDE
% Predicados para a fórmula resolvente
delta(A, B, C, D):- D is B*B - 4*A*C.
equation(A,B,C,R1,R2):-delta(A,B,C,D),R1 is (-B+sqrt(D))/(2*A), R2 is (-B-sqrt(D))/(2*A).

% Criação de diálogo sobre a Fórmula resolvente
:-create_dialog('Fórmula Resolvente').
:-add_numForm_dialog('Fórmula Resolvente','a').
:-add_numForm_dialog('Fórmula Resolvente','b').
:-add_numForm_dialog('Fórmula Resolvente','c').
:-dialog_setAction('Fórmula Resolvente','getNumField("Fórmula
Resolvente","a",A),getNumField("Fórmula Resolvente","b",B),getNumField("Fórmula
Resolvente","c",C),equation(A,B,C,R1,R2),dialog_setOutput("Fórmula Resolvente",[R1,R2])').

% Criar menu com um item que evoca a fórmula resolvente
:-create_menu('Math Menu','Quadratic Equation','show_dialog("Fórmula Resolvente")').

% Criar um diálogo para testes
:-create_dialog('Teste').
:-add_textForm_dialog('Teste','Msg').
:-
dialog_setAction('Teste','getTextField("Teste","Msg",Output),writeln(Output),dialog_setOutput("Teste
",[Output])').

%Criar menu com um item que possibilita abrir um ficheiro externo no editor
:-create_menu('Cx
File',['Open','file_chooser(Path),open_file(Path)','Consult','editor_getCurrOpenFilePath(Path),consult(
Path)']).
Adicionar item CxProlog no ContextMenu do editor CxProlog com alguns sub-itens
:-add_contextItem('Selected Text','selectedText(Text)').
:-add_contextItem('Inject selected text','editor_injectSelCode').
```

```
%Definir regras para syntax highlight
:-editor_singleLineRule('word '','"', '"', '\\', 100, 100, 100).
:-editor_singleLineRule('word '','"', '"', '\\', 50, 50, 50).
:-editor_endLineRule('comment %','%', 150, 0, 150).
:-editor_varRule('variables',0,255,0).
:-editor_addNormalTextRule('default text',0,0,0).
:-editor_addPropertyHighlight('built_in',0,0,255).
:-editor_addPropertyHighlight('user_defined',0,255,0).

:-editor_setContentAssistFile('/src/builts.xml').
:-getUserDefined(_).
```